MASTER EN SCIENCES INFORMATIQUES

Un Nouvel Algorithme de Calcul de l'Ensemble des États Accessibles d'un Système d'Automates Communicants

CASTIAUX, Jean-Christophe

Award date:
1995

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix Namur
Institut d'Informatique

Un Nouvel Algorithme de Calcul de l'Ensemble des Etats Accessibles d'un Système d'Automates Communicants

Jean-Christophe Castiaux

Promoteur : Professeur B. Le Charlier
Co-promoteur : D. Zampuniéris

Année Académique 1994-1995

Un grand merci à Denis Zampuniéris pour sa disponibilité et ses conseils tout au long de ce mémoire, à mon promoteur le Professeur Baudouin Le Charlier pour ce mémoire qu'il m'a permis de réaliser et à tous ceux, famille et amis, qui durant cette année ont été un soutien moral non négligeable.

Table des matières

Résumé

Il a été montré récemment qu'il est désormais possible de calculer les états accessibles d'automates synchronisés de manière efficace, aussi bien du point de vue du temps de calcul que du stockage des données. Nous exposons dans ce mémoire un nouvel algorithme de calcul de l'ensemble des états accessibles d'automates synchronisés. Celui-ci est basé sur une structure de données compacte de stockage de relations n-aires, appelée "arbre partagé". Nous montrons comment notre algorithme peut être optimisé des points de vue temps d'exécution et espace de stockage. Nous comparons nos résultats avec ceux obtenu par d'autres algorithmes connus.

Abstract

It has been shown recently that computing the set of accessible states of synchronized automatas is now possible, considering both computing time and data storage. We present a new algorithm computing the set of accessible states of synchronized automatas. It is based on a compact data structure designed for the storage of n-ary relations and called "Sharing Tree". We show that the algorithm can be optimized concerning both execution time and storage space. We compare our results with those obtained by other well known algorithms.

Introduction

La connaissance de l'ensemble des états accessibles d'un système d'automates synchronisés est un outil très utile pour la mesure des performances d'un tel système. Elle permet entre autres de déterminer si l'interblocage des différents automates est possible ou probable ou encore d'évaluer les états qui, pour chaque automate, seront les plus visités. Les diverses informations permettent de déduire certaines propriétés des automates qui précisent leur mode de fonctionnement.

Le calcul des états accessibles d'un ensemble de programmes parallèles est un problème difficile. En effet, le nombre d'états du sytème parallèle croît généralement de manière exponentielle avec le nombre de programmes. Dès lors, l'utilisation de méthodes énumératives pour le stockage ou le calcul de ces états est tout à fait exclue pour les cas non triviaux puisque ces méthodes débouchent sur des algorithmes exponentiels en place et en temps.

Ce problème a longtemps constitué une barrière à la progression des recherches sur le parallélisme. Il a amené divers chercheurs à concevoir des structures de données permettant de représenter des ensembles de n-uplets de manière compacte et à manipuler efficacement ces ensembles (BDD's, ...).

Les BDD's [6] [5] se présentent encore actuellement comme l'outil de référence lorsqu'on parle de structure de données adaptées aux stockages des n-uplets et à leur manipulation. Cependant, une nouvelle structure de donnée a été mise au point par Denis Zampunieris et Baudouin Le Charlier [2]. Cette structure est appelée Arbre Partagé (Sharing Tree).

L'avantage des Arbres Partagés est d'être particulièrement bien adaptés aux traitements sur les automates en parallèles. En effet, ils permettent une décomposition de la structure en niveaux de sorte que chaque niveau correspondent aux états d'un automate donné. De plus, cette structure de stockage compacte permet de traiter la majorité des opérations ensemblistes sur les n-uplets sans aucun décompactage et de manière rapide (union, intersection, soustraction, inclusion, ...)

Des recherches ont déjà été réalisées et ont permis de démontrer l'efficacité de cette structure dans la recherche des états accessibles du produit synchronisé d'un système d'automates [1]. Dans un premier temps, l'algorithme mis au point (DZA&BLE) présentait des temps exponentiels en fonction du nombre d'automates du système. Cependant, grâce à une technique d'optimisation basée sur un caching efficace des opérations effectuées, il a été possible de réduire ces temps à des temps linéaires. Ces derniers mesurés sur les benchmarks habituels dépassaient tous les résultats connus à ce jour.

Nous avons décidé de réaliser un nouvel algorithme de calcul de l'ensemble des états accessibles du produit synchronisé d'un système d'automates. Cet algorithme est également basé sur la structure d'Arbre Partagé. Il nous a paru intéressant d'évaluer sur cette même structure un algorithme utilisant une technique de recherche entièrement différente de celle de DZA&BLE. En effet, notre algorithme se sert d'une stratégie de stabilisation locale par niveaux alors que DZA&BLE utilise une méthode de calcul global de point fixe. Nous espérons qu'une stratégie locale permettra, par le traitement de plus petits ensembles, d'obtenir de meilleurs résultats que DZA&BLE.

Dans le premier chapitre, nous exposerons la théorie sur laquelle est basée notre recherche : les automates finis qui sont la base même de ce mémoire et les arbres partagés qui constitue l'outil principal permettant d'obtenir des algorithmes performants.

Ensuite, dans le deuxième chapitre, nous analyserons l'algorithme DZA&BLE sur lequel seront basées la plupart de nos comparaisons. Nous verrons comment, par un caching efficace, il a été possible de transformer un algorithme aux temps a priori exponentiels en un algorithme linéaire.

Le troisième chapitre constitue la partie la plus importante de ce mémoire. Nous y développons un nouvel algorithme de calcul des états accessibles et montrons comment il a été possible de l'implémenter dans le même environnement que DZA&BLE. Ensuite nous comparons les temps d'exécution de notre algorithme avec ceux l'algorithme originel et analysons les différentes optimisations qu'il nous a été possible de faire. En fin de chapitre, nous faisons également une critique de l'implémentation et donnons des indices sur les différentes alternatives d'implémentations qui existent ainsi que les conséquences possibles d'une implémentation différente sur les performances de l'algorithme.

Le chapitre 4 est un récapitulatif de tous les résultats obtenus. Il est destiné à fournir au lecteur une vue globale de ce qui a été réalisé dans ce mémoire et permet d'effectuer en conclusion le bilan des recherches effectuées ainsi que les possibilités d'extension et d'avenir de ce mémoire.

Chapitre 1.
Automates à états finis et application à l'étude du parallélisme

1. Les automates à états finis

1.1. Définitions

Définissons tout d'abord l'ensemble $\mathcal{A}$, ensemble d'actions, avec $\mathcal{A} = A \cup \overline{A} \cup \{\tau\}$ où $a \in A$ et $\overline{a} \in \overline{A}$ sont appelées actions complémentaires.

Un automate [8] est défini par le tuple < S, i, T > où

- S est un ensemble fini, appelé l'ensemble des états.
- $i \in S$ est l'état initial.
- $T \subseteq S \times \mathcal{A} \times S$ est un ensemble de transitions; (s, a, s') sera noté $s \xrightarrow{a} s'$.

Soit une transition t = (s, a, s'). On parlera parfois de manière abusive d'une transition complémentaire à t et on notera $\overline{t}$ une transition (s'', $\overline{a}$, s'''), où $\overline{a}$ est l'action compémentaire à a.

1.2. Représentation graphique

On représente un automate par un graphe étiqueté, où l'état initial est indiqué par une flèche. Chaque état de S est donc représenté par un noeud du graphe alors qu'une transition $s \xrightarrow{a} s'$ est représentée par une transition du graphe entre les noeuds s et s', étiquetée par a.

1.3. Etats accessibles

Soit un automate Aut = <S, i, T>.
On dit que l'état $s \in S$ est accessible si et seulement si il existe une suite de transition $(t_i)_{i \in [1..n]}$ et une suite d'états $(s_i)_{i \in [0..n]}$ telles que

- $t_i = (s_{i-1}, a_i, s_i)$ $i = 1, \ldots, n$
- $s_0 = i$
- $s_n = s$

Un état s est donc accessible si, en partant de l'état initial de l'automate, on peut trouver une suite de transitions qui mène à s.

On définit encore l'ensemble des états accessibles d'un automate Aut = <S, i, T> comme l'ensemble des états s ∈ S accessibles depuis i.

1.4. Exemple

Soit l'automate **Aut1** défini comme suit :

Aut1 = <S, i, T> où
S = {A, B, C, D, E, F},
i = A
et T = {(A, t1, B), (A, τ, B), (A, τ, C), (B, t2, D), (C, τ, D), (E, t3, F)}

Cet automate comporte quatre états, deux transitions dites synchrones (t1 et t2) et trois transitions asynchrones ou internes. L'état initial est A.

On peut représenter l'automate **Aut1** par le graphe représenté Figure 1 :

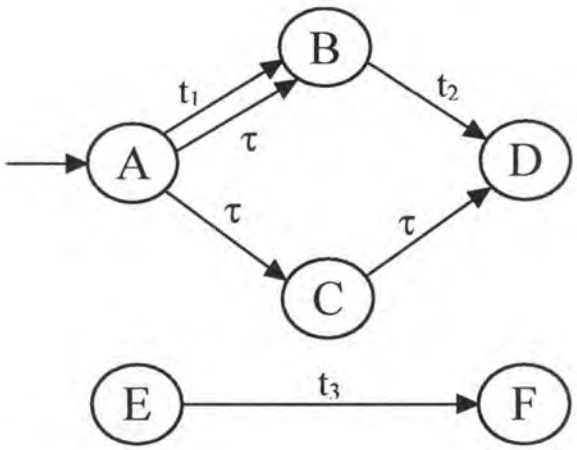

Figure 1 : Représentation graphique de l'automate Aut1

L'ensemble des états accessibles de l'automate **Aut1**, noté Access(**Aut1**) est le suivant :

Access(**Aut1**) = {A, B, C, D}

Les états E et F ne sont pas accessibles. En effet, aucune suite de transitions partant de l'état initial A n'arrive en E ou F.

2. Automate produit

2.1. La mise en parallèle de plusieurs automates

La mise en parallèle de plusieurs automates consiste à considérer ces automates non plus comme des entités autonomes mais comme faisant partie d'un système d'automates qui interagissent.

On introduit pour cela un point de vue dynamique de la notion d'automate. Pour ce faire, on définit à tout instant l'état courant dans lequel l'automate se trouve. A l'instant initial, l'état courant est l'état initial pour tous les automates. Ensuite, cet état courant évolue lorsque l'automate a la possibilité d'effectuer une transition vers un autre état.

Les interactions constituent des points de synchronisation. Un automate ne pourra effectuer une transition t que si l'état courant d'un autre automate permet d'effectuer la transition complémentaire . Ceci implique des attentes et la connaissance de l'état courant des autres automates.

Ce sont les transitions synchrones qui déterminent les synchronisations entre automates.

2.2. Transitions synchrones et asynchrones

Les transitions asynchrones sont des transitions internes à un automate. Elles n'impliquent aucune synchronisation. L'action correspondant à une transition asynchrone est l'action τ. Elle représente toute action interne à un automate.

Les transitions synchrones au contraire impliquent une synchronisation. On définit la synchronisation de deux automates $Aut_j = <S_j, i_j, T_j>$ et $Aut_k = <S_k, i_k, T_k>$ d'un système d'automates $Aut_l = <S_l, i_l, T_l>$ (l = 1..n) comme suit :

Soient deux transitions $t_j \in T_j$ de Aut_j et $t_k \in T_k$ de Aut_k. Alors ces transitions déterminent une synchronisation possible de Aut_j et Aut_k si et seulement si elles sont complémentaires c'est-à-dire si les actions sous-jacentes sont complémentaires (cfr. 1.1).

Dans la suite nous considérerons qu'une transition d'un automate donné ne peut être synchrone qu'avec un et un seul autre automate du système. Ainsi, lorsqu'on considérera une transition synchrone dans un automate Aut_i, on supposera connu l'automate Aut_j contenant la ou les transitions synchrones "complémentaires" à celle-ci (c'est-à-dire les transitions de Aut_j correspondant à des actions complémentaires à l'action représentée par la transition de Aut_i).

2.3. Automate produit

Soit $Aut_j = < S_j, i_j, T_j >$ (j = 1 .. n) un système d'automates.
Le produit synchronisé du système est l'automate PSync définit (cfr. [9]) par le tuple $< S, I, T >$ où :

- $S = S_1 \times S_2 \times ... \times S_n$
- $I = < i_1, ..., i_n >$
- $T \subseteq S \times S$ est l'ensemble des couple ($< s_1, ..., s_n >, < s'_1, ..., s'_n >$) tels que

$$\text{soit } \exists i \in 1...n \text{ t.q. } \left[\forall k \neq i \quad s_k = s_k' \wedge \exists t \in T_i \text{ t.q. } t = s_i \xrightarrow{\tau} s_i' \right]$$

$$\text{soit } \exists i, j \in 1...n \text{ t.q. } \left[\forall k \neq i, j \quad s_k = s_k' \wedge \exists t \in T_i, t' \in T_j \text{ t.q. } \begin{cases} \bullet\ t = \bar{t}' \\ \bullet\ s_i \xrightarrow{t} s_i' \\ \bullet\ s_j \xrightarrow{t'} s_j' \end{cases} \right]$$

L'automate représentant le produit synchronisé est appelé automate produit du système d'automates Aut_j (j = 1 .. n).

Un état de l'automate produit sera donc un tuple formé des n états des automates composants. De même, une transition de l'automate produit sera soit une transition interne à un des automates (c.-à-d. un seul des états aura changé), soit une transition synchrone entre deux automates (c.-à-d. deux états auront changé grâce à l'application de deux transitions complémentaires).

L'automate produit représente donc l'ensemble des possibilités d'exécution des n automates du système en parallèle.

2.4. Exemple

Soit le système d'automates formé par les automates **Aut1** de la Figure 1 et **Aut2** de la Figure 2. Alors le produit synchronisé de ces deux automates est définit par l'automate PSync représenté graphiquement à la Figure 3.

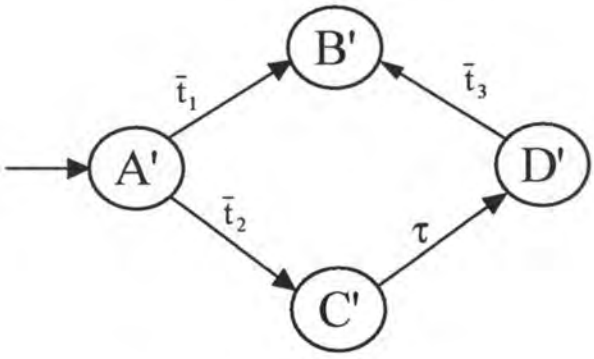

Figure 2 : Graphe représentant l'automate Aut2

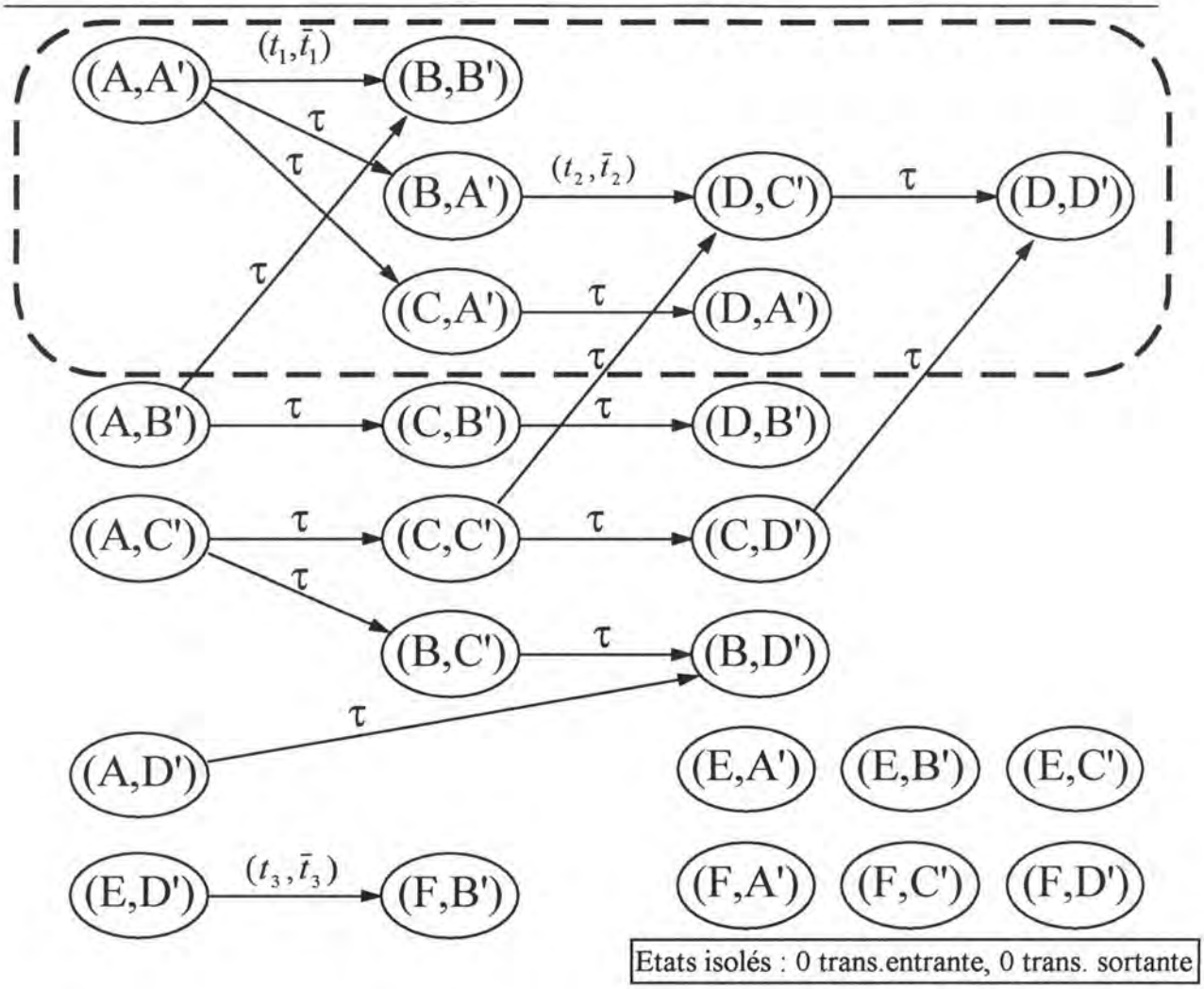

Figure 3 : Graphe de l'automate produit de Aut1 et Aut2

On peut distinguer facilement les transitions internes à chaque automate (représentées par τ) et les transitions synchrones complémentaires (représentée par $(t, \bar{t})$).
On peut déjà remarquer que la taille de l'automate produit est de loin supérieure à celle des automates de départ.

2.5. Etats accessibles du produit synchronisé d'un système d'automates

Soit un système d'automates Aut_j, = < S_j, i_j, T_j > j = 1 .. n et soit PSync = < S, I, T > l'automate produit de ce système.

Alors, un état accessible du produit synchronisé de ce système est un état accessible de l'automate PSync (cfr 1.3.). En effet, nous avons vu que l'automate produit représentait l'ensemble des possibilités d'action des automates du système en parallèle.

De même, l'ensemble des états accessibles est donc l'ensemble des $s \in S$ accessibles grâce à une suite de transition $(t_i)_{i=1..m}$ issue de I.

D'un point de vue intuitif, un état accessible du produit synchronisé et un tuple d'états représentant une configuration possible des états courants du système d'automates à un moment donné.

Dans l'exemple Figure 3, l'ensemble des états accessibles de PSync, produit synchronisé de Aut1 et Aut2, a été encadré en trait discontinu.

2.6. Croissance exponentielle

On peut calculer facilement le nombre d'états du produit synchronisé d'un système d'automates. En effet, comme l'ensemble des états du produit synchronisé est le produit cartésien des ensembles d'états des automates composants, sa cardinalité est le produit des cardinalités de chaque ensemble d'états. On peut donc en déduire que le nombre d'états du produit synchronisé varie de manière exponentielle avec le nombre d'automates composant le système.

En ce qui concerne l'ensemble des états accessibles, on ne peut pas faire un raisonnement aussi formel. Effectivement, cet ensemble dépend essentiellement de la structure des automates impliqués. Plus les combinaisons de transitions possibles seront variées, plus l'ensemble accessible sera grand. Cependant, l'expérience a montré que pour les automates rencontrés, le nombre d'états accessibles du produit synchronisé varie également de manière exponentielle.

Ces considérations nous amène à aborder le problème du temps de calcul de l'ensemble des états accessibles d'un système d'automates. Toute méthode énumérative de calcul de cet ensemble aura un temps de calcul dépendant exponentiellement du nombre d'automates. Il est dès lors impensable d'utiliser une telle méthode pour un calcul efficace de cet ensemble. C'est pourquoi des algorithmes de calcul plus performants ont été mis au point, utilisant des structures de données adaptées et permettant des temps de calcul beaucoup plus acceptables.

De même le stockage des données occupe lui aussi un espace croissant de manière exponentielle. Il est donc indispensable de construire des structures de données qui combinent économie de l'espace de stockage et performance au niveau du temps de calcul.

2.7. Utilité et applications

2.7.1. La vérification

Les automates sont souvent utilisés pour modéliser des applications foncionnant en parallèle. C'est le cas des programmes parallèles (avec machine multiprocesseur), des systèmes distribués, des programmes d'accès à une base de donnée.

L'automate produit permet la détection de problèmes tels que l'interblocage des deux applications. En effet, un état représente un interblocage possible s'il fait partie de l'ensemble

accessible de l'automate produit et s'il n'existe aucune transition partant de cet état vers un autre état de l'automate produit.

De plus, il est souvent intéressant de pouvoir déterminer si un état fait partie de l'ensemble des états accessibles (par exemple un état terminal). Là aussi le calcul de l'automate produit ou même "simplement" de l'ensemble des états accessibles va permettre de répondre à ce genre de question.

2.7.2. Des statistiques

Il peut être utile dans certaines applications parallèles de déterminer si un automate du système se trouve souvent dans un état donné. Ceci peut être fait grâce à l'ensemble des états accessibles. Plus généralement, il est souvent nécessaire de vérifier certaines propriétés utiles des états accessibles au moyen de statistiques.

Par exemple, on peut déduire de l'ensemble des états accessibles de PSync (produit synchronisé de Aut1 et Aut2) que 4 états sur 7 ont pour deuxième composant A'. Ceci pourrait être une information importante dans l'analyse du comportement de l'automate Aut2 lorsqu'il fonctionne en parallèle avec Aut1.

2.7.3. Conclusion

L'ensemble des états accessibles d'un système d'automate peut donc être un outil très utile dans l'analyse d'applications parallèles. Connaître l'automate produit complètement (c.-à-d. les états et les transitions correspondantes) serait évidemment plus intéressant dans le sens où l'on pourrait analyser de manière plus fine le comportement des applications. Cependant, il est difficile (surtout pour des raisons de stockage) de construire complètement cet automate. C'est pourquoi généralement, on se satisfait de l'ensemble des états accessibles qui permet de déterminer certaines propriétés importantes dans les systèmes parallèles.

3. Une structure de stockage pour la manipulation de relations n-aires (tuples) de manière compacte : Les Arbres Partagés

3.1. Définition d'un Arbre Partagé (AP)

Un arbre partagé [2] est une structure de données qui représente un ensemble de tuples (relations n-aires). Cette structure de données a la forme d'un graphe sans cycle et possédant une racine.

On définit de manière formelle un arbre partagé par le quadruplet (N, V, val, succ) où

- $N = N_0 + N_1 + ... + N_k$ $(k \geq 0)$ est l'ensemble des noeuds de l'arbre organisé en couches (N_i formant l'ensemble des noeuds de la couche i)
- $val : N \rightarrow V + \{\bot, \top\}$ est la fonction qui associe une valeur à un noeud.
- $succ : N \rightarrow \wp(N)$ donne l'ensemble des successeurs d'un noeud.

et où les règles suivantes sont vérifiées :

1. $\forall\, 0 \leq i \leq k, \quad \forall\, n \in N_i, \quad succ(n) \subseteq N_{i+1}$
   Chaque noeud a ses successeurs dans la couche suivante.

2. $\forall\, 0 \leq i \leq k, \quad \forall\, n_1, n_2 \in N_i, \quad n_1 \neq n_2 \quad val(n_1) = val(n_2) \Rightarrow succ(n_1) \neq succ(n_2)$
   Deux noeuds égaux dans la même couche n'ont pas même ensemble de fils.

3. $\forall\, n \in N, \quad \forall\, s_1, s_2 \in N_i, \quad s_1 \neq s_2 \Rightarrow val(s_1) \neq val(s_2)$
   Un noeud n'a pas deux fils de même valeur.

4. $\#N_0 = 1 \quad \wedge \quad \forall\, n \in N, val(n) = \top \Leftrightarrow n \in N_0$
   La première couche contient un seul élément de valeur "T" appelé racine de l'arbre partagé.

5. $val(n) = \bot \Rightarrow succ(n) = \varnothing \quad \wedge \quad succ(n) = \varnothing \Rightarrow (val(n) = \bot \vee val(n) = \top)$
   Un noeud final a toujours pour valeur "⊥" sauf s'il est aussi la racine, auquel cas il a la valeur "T".

3.2. Eléments d'un arbre partagé

Un arbre partagé AP = (N, V, val, succ) représente l'ensemble $\mathcal{Elem}$(AP) des tuples présents sur les chemins partant de la racine $r \in N_0$.

Plus formellement,

$\mathcal{Elem}(\mathrm{AP}) = \mathrm{Set(r)}$

$$\text{où } \forall n \in N \quad Set(n) = \begin{cases} \{()\} & \text{si val(n)} = \bot \\ \bigcup_{s \in succ(n)} Set(s) & \text{si val(n)} = \mathrm{T} \\ \{val(n)\} \times \bigcup_{s \in succ(n)} Set(s) & \text{sinon} \end{cases}$$

3.3. Exemple graphique

Comme nous l'avons vu ci-dessus, un arbre partagé a la forme d'un graphe acyclique possédant une racine.

La Figure 4 nous montre un exemple d'arbre partagé. La fonction succ est donc représentée par un lien entre deux niveaux de l'arbre alors que la fonction val donne une valeur à chaque noeud du graphe.

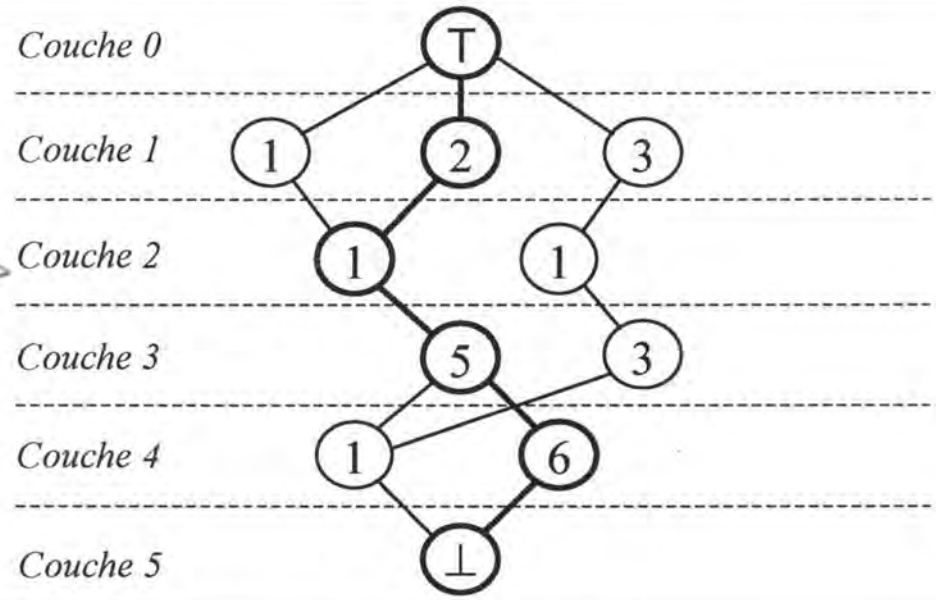

Figure 4 : Graphe représentant un arbre partagé AP

Cet arbre partagé AP représente l'ensemble de tuples suivant :

$\mathcal{Elem}(\mathrm{AP}) = \{(1, 1, 5, 1), (1, 1, 5, 6), (2, 1, 5, 1), (2, 1, 5, 6), (3, 1, 3, 1)\}$

En gras sur le schéma, le chemin parcouru pour détecter l'élément (2, 1, 5, 6)

3.4. Le concept de Arbre Semi-Partagé (ASP)

Souvent il est plus commode de considérer la structure d'arbre partagé sans tenir compte de la deuxième règle. En effet, la vérification de cette règle est difficile et implique souvent de longs calculs notamment lors de l'ajout d'un élément ou d'un noeud à l'arbre partagé.

Cette difficulté a amené les chercheurs concernés à utiliser une structure plus souple appelée Arbre Semi-Partagé (ASP) [2]. Cette structure est identique à celle d'arbre partagé, sauf que l'on omet la règle n° 2 (c.-à-d. deux noeuds de même valeur sur la même couche peuvent avoir même ensemble de fils). Une telle structure est plus souple et permet des mises à jour faciles.

Il existe en outre un algorithme de réduction d'un ASP en AP de complexité $0(\#\text{noeuds})^2$.

L'inconvénient des ASPs par rapport aux APs est qu'ils occupent plus d'espace. En effet, l'arbre comporte alors des structures redondantes puisque deux noeuds de même valeur peuvent avoir même ensemble de fils. Cette règle n'implique pas seulement la duplication des noeuds en question, mais peut entraîner, récursivement la duplication de grandes parties de l'arbre. La Figure 5 nous montre une version catastrophique d'un ASP représentant le même ensemble d'éléments que l'AP Figure 4. On voit que le nombre de noeuds a augmenté considérablement. Dans cet exemple, toute une partie de l'arbre a été dupliquée. Notons que dans certains cas, une telle partie pourrait être dupliquée plus que deux fois, entraînant alors une saturation de l'espace mémoire disponible.

Cette faiblesse des ASPs nous obligera dans nos algorithmes à effectuer régulièrement une "réduction" de l'arbre en construction. Nous expliquerons en détails dans le chapitre 3 un algorithme de réduction des ASPs fonctionnant en $0(\#\text{noeuds})^2$.

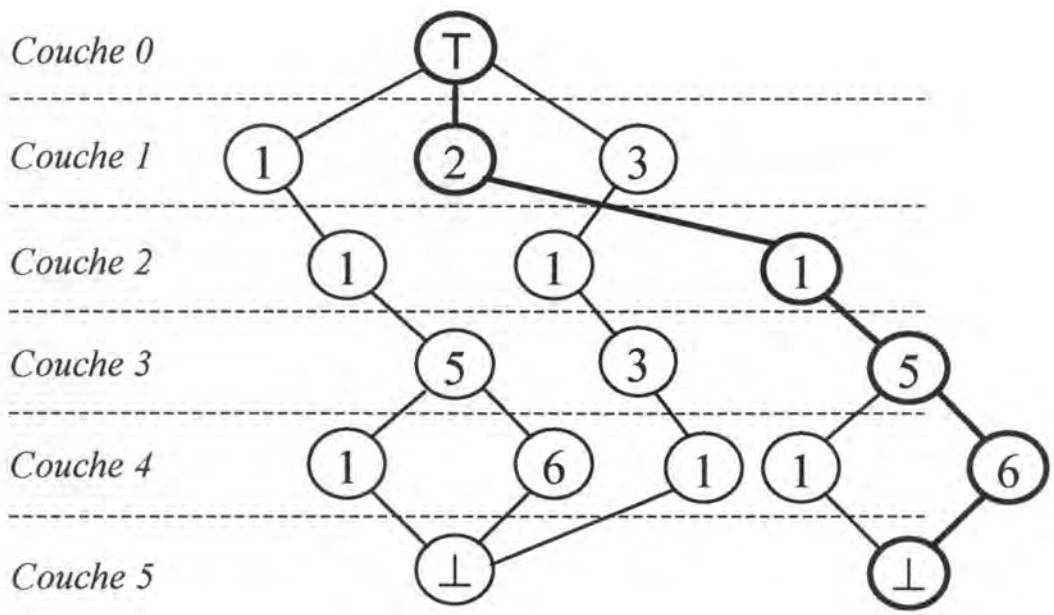

Figure 5 : Version semi-partagée de l'AP de la Figure 4

3.5. Avantages de la structure

3.5.1. Point de vue statique

Cette structure permet le stockage de tuples de manière efficace. Le compactage est d'autant plus grand que la cardinalité des ensembles de valeurs de chaque couche est petite. En effet, la structure d'AP utilise les redondances au niveau de chaque couche pour épargner de l'espace.

S'il n'y a que très peu de redondances, la structure est inefficace. Par contre, si les ensembles de valeurs ont des cardinalités restreintes, les redondances seront exploitées au mieux par la structure et on pourra, avec un très petit nombre de noeuds, représenter un très grand nombre de tuples.

3.5.2. Point de vue dynamique

La structure d'arbre partagé comporte également l'avantage d'une panoplie d'algorithmes de manipulation de complexité 0(#noeuds). Ces algorithmes permettent des transformations rapides d'AP (ou de ASP), ce qui autorise leur utilisation dans des algorithmes de recherche plus complexes.

On peut, par exemple, facilement ajouter ou supprimer une couche, un noeud, un fils, vérifier si un noeud possède un fils de valeur donnée, si un arbre possède un élément donné, et effectuer la majorité des opérations ensemblistes (union, intersection, soustraction, ...) sur les APs ou les ASPs.

3.6. Utilisation des arbres partagés pour le stockage des états accessibles d'un automate produit

Nous avons vu que la structure d'arbre partagé était adéquate lorsque les cardinalités des ensembles de valeurs de chaque couche étaient relativement petites. Or c'est le cas pour les états d'un automate produit. En effet, la cardinalité de chaque composant d'un état est fonction de celle de l'ensemble des états de l'automate du système correspondant (assez restreinte en général).

De plus, la structure d'arbre partagé s'adapte particulièrement bien au problème du calcul des états accessibles de l'automate produit synchronisé. En effet, on peut considérer lors du calcul que chaque couche de l'AP représente les états de l'automate correspondant déjà visités. Dès lors, étant donné un AP partiellement construit comportant déjà tous les états trouvés, il suffit pour trouver des états supplémentaires de considérer chaque noeud et de voir s'il est possible d'effectuer à partir de celui-ci une transition synchrone ou asynchrone.

Dans le cas d'une transition asynchrone, on peut facilement ajouter un fils de valeur la destination de la transition au père du noeud de valeur la source de la transition (si ce noeud existe déjà, il faudra procéder de manière sensiblement différente). Le cas des transitions synchrones est plus compliqué, mais assez similaire dans l'esprit. Nous exposerons dans les

chapitres suivants des algorithmes de calcul qui tirent leurs performances de la structure de données considérée.

On peut encore signaler que le taux de réduction de l'espace de stockage n'est évidemment pas toujours le même. On remarque qu'il est fonction du taux d'interdépendance (de synchronisation) des automates. Si les automates sont très fortement synchronisés (beaucoup de transitions synchrones, peu de transitions asynchrones), le taux de réduction est assez restreint alors que, dans le cas d'automates asynchrones, ce taux est au contraire très élevé. Cependant, la structure est dans la grande majorité des cas beaucoup plus performante que le stockage naturel des tuples (tableaux ou listes).

Chapitre 2
Calcul des états accessibles d'un automate produit

1. Introduction

Comme nous l'avons entrevu précédemment, un algorithme performant pour le calcul des états accessibles d'un système d'automates doit utiliser des techniques permettant de découvrir le plus d'états accessibles par étape. Il n'est pas question d'énumérer tous les états accessibles. Pour ce faire, il est nécessaire d'utiliser une structure de données adéquate. Dans ce chapitre nous examinons un algorithme de calcul de ces états basé sur la structure de manipulation de données applelée Arbre Partagé. Cet algorithme utilise une technique de calcul global de point fixe sur l'ensemble des états accédés. Nous verrons dans le chapitre suivant un autre algorithme mis au point lors de la réalisation de ce mémoire et utilisant une stratégie de stabilisation locale par niveau.

Dans la première partie de ce chapitre, nous étudions le fonctionnement de l'algorithme DZA&BLE [1]. Ensuite, nous voyons les optimisations effectuées et nous terminons par une analyse des résultats obtenus et une comparaison avec les algorithmes les plus connus.

2. Algorithme DZA&BLE

2.1. Moyens utilisés : Les arbres partagés

L'algorithme utilise la structure d'arbre partagé pour le stockage des états accédés lors de la recherche. Cette structure a été enrichie pour que chaque noeud contienne l'information nécessaire à sa manipulation. La structure d'un noeud d'un arbre partagé a donc été définie comme suit :

```
TNode = record
      ident       : integer;      {L'identifiant du noeud dans l'arbre}
      value       : state;        {La valeur du noeud : un état d'un automate}
      succ        : TSetOfNodes;  {Les successeurs du noeuds}
      nbFathers   : integer;      {Le nombre de pères du noeuds}
      nbElem      : integer;      {Nb de tuples dans le ss-AP de racine le noeud}
      result      : TNode;        {Paramètre utilisé par l'algorithme}
      marked      : Boolean;      {Paramètre utilisé par l'algorithme}
end;
```

Comme nous l'avons vu précédemment, il est souvent plus commode dans les algorithmes d'utiliser une structure d'arbre sous-partagé (ASP), cette structure permettant des insertions rapides dans l'arbre. C'est ce qui est réalisé par l'algorithme, tout en réduisant après chaque itération l'arbre sous-partagé en arbre partagé.

Nous considérons maintenant le système d'automates $Aut_j = < S_j, i_j, T_j >$ (j = 1 .. n). Le calcul de l'ensemble des états accessibles de l'automate produit consiste à construire l'arbre semi-partagé PSync = < N, V, val, succ >, ou la fonction *val* associe à chaque noeud *N* une valeur *V* qui est l'état représenté par ce noeud.

Les procédures de manipulation de données utilisées par l'algorithme sont les suivantes :

- **AddNode** : Int × V → Nodes
  AddNode(i,v) ajoute à la couche *i* de l'arbre sous-partagé PSync un noeud de valeur *v*.

- **AddSucc** : N × N → N
  AddSucc(n, n') ajoute *n'* à l'ensemble des successeurs de *n*.

- **AddGraph** : N × N → N
  AddGraph(n, n') ajoute l'arbre sous-partagé de racine *n'* (noté $ASP_{n'}$) l'ensemble des successeurs de *n*. C'est-à-dire si E = $\mathcal{Elem}(ASP_n)$, alors, après l'exécution de *AddGraph(n, n')*, on aura $\mathcal{Elem}(ASP_n) = E \cup \{< n, \alpha > : \alpha \in \mathcal{Elem}(ASP_n)\}$

- **RemoveMarks**( ) change la valeur du champ *marked* de tous les noeuds de l'arbre semi-partagé PSync à false.

2.2. Analyse du fonctionnement de l'algorithme

2.2.1. Fonctionnement de l'algorithme

Pour mieux comprendre comment l'algorithme procède, nous allons donner une défintition formelle de l'ensemble des états accessibles d'un produit synchronisé sous la forme du point fixe d'une transformation.

Considérons $Aut_j = < S_j, i_j, T_j >$ (j = 1 .. n), un système d'automates et PSync = < S, I, T >, l'automate produit correspondant. Alors on définit l'ensemble des états accessibles du produit synchronisé comme le plus petit point fixe Y* de la transformation

$$Acces : P(S) \rightarrow P(S)$$
$$Acces(Y) = Y \cup (\bigcup_{j=1..n} synchro_j(Y)) \cup (\bigcup_{j=1..n} self_j(Y))$$

où :

- $self_j(Y)$ est l'ensemble des états accessibles à partir des états de Y en effectuant toutes les transitions internes (c.-à-d. asynchrones) possibles dans l'automate j.
  $$self_j(Y) = \{\langle s_1,\ldots,s_j',\ldots,s_n\rangle : \langle s_1,\ldots,s_j,\ldots,s_n\rangle \in Y \wedge s_j \xrightarrow{\tau} s_j' \in T_j\}$$

- synchro$_j$(Y) est l'ensemble des états accessibles à partir des états de Y en effectuant toutes les transitions synchrones possibles entre l'automate j et les autres.

$$synchro_j(Y) = \left\{ \left\langle s_1,\ldots,s_j',\ldots,s_k',\ldots,s_n \right\rangle : \left\langle s_1,\ldots,s_j,\ldots,s_k',\ldots,s_n \right\rangle \in Y \wedge s_j \xrightarrow{a} s_j' \in T_j \wedge s_k \xrightarrow{\bar{a}} s_k' \in T_k \right\}$$

On peut faire deux observations sur la fonction Access et sur l'ensemble accédé. Tout d'abord la fonction Access vérifie la propriété suivante

$$\forall Y \quad Y \subseteq Acces(Y)$$

De plus, si Y = Acces(Y), alors Y est un point fixe de la transformation. Ceci implique qu'on peut obtenir ce point fixe en appliquant de manière itérative la transformation Y en commençant par Y = {I}.

On peut aussi remarquer que cette technique implique un recalcul de la totalité des éléments lors de chaque itération. Nous verrons plus loin qu'il existe des méthodes permettant d'éviter ce gaspillage de temps en enregistrant les opérations déjà effectuées.

La Figure 6 nous montre un exemple du calcul des états accessibles grâce à la théorie du point fixe. L'exemple choisi est l'automate produit du chapitre précédent. Chaque ensemble de noeuds dessiné (grisé du plus clair au plus foncé) montre l'ensemble des états accédés à cette itération. On voit que les différents ensembles tendent vers l'ensemble des états accessibles et aussi qu'il y a recalcul de toutes les transitions à chaque itération.

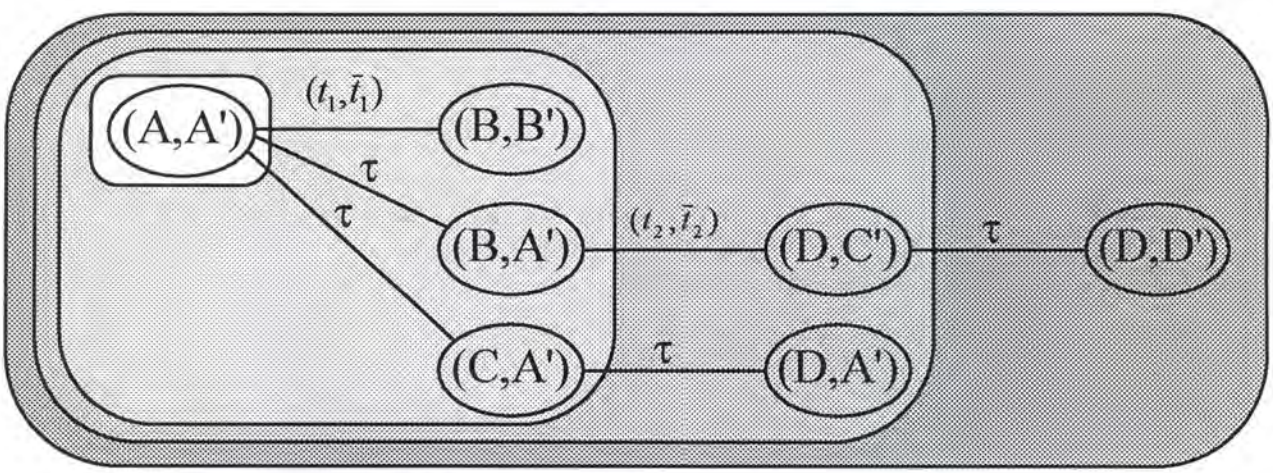

Figure 6 : Exemple du calcul des états accessibles par la méthode du point fixe.

L'algorithme que nous exposons dans ce chapitre va utiliser cette propriété de l'ensemble des états accessibles pour construire celui-ci de manière itérative. De manière schématique, le programme aura la forme suivante :

```
procedure compute_fixpoint(out Y : set of states);

begin
  Y := {I};
  while Y is enlarged do
    Y := Y U self_1(Y);
```

```
        ...
        Y := Y U self_n(Y);
        Y := Y U synchro_1(Y);
        ...
        Y := Y U synchro_n(Y);
      end{while}
    end
```

2.2.2. *Structure de l'algorithme*

L'algorithme se présente sous la forme de deux procédures principales.

La première permet de calculer le résultat de l'application de toutes les transitions asynchrones d'un automate du système quand elles sont appliquées aux états de cet automate déjà visité. Elle est détaillée à la Figure 7.

La seconde traite le cas des transitions synchrones. A partir d'un noeud donné, on recherche s'il est possible d'effectuer une ou plusieurs des transitions synchrones impliquant l'état correspondant à ce noeud. Ce traitement est effectué pour tous les noeuds de l'arbre. Cette procédure est détaillée à la Figure 8.

```
procedure Self(in j:integer);

var
  s,s':state;
  n,n',n",newnode:TNode;

begin
  forall s ∈ s_j do
    forall s --τ--> s' ∈ T_j do begin
      forall n ∈ N_j-1 do
        forall n' ∈ n.succ do begin
          if not n'.marked
          then begin
            n'.marked := true; % marque le noeud comme "visité"
            if n'.value = s
            then begin          % application d'une transition
              newnode := AddNode(j,s');
              forall n" ∈ n'.succ do
                AddSucc(newnode,n");
              n'.result := newnode;
            end
          end;
          % On ajoute au père de n' les éléments trouvé en effectuant la transition
          if n'.result ≠ null
          then AddGraph(n,n'.result)
        end;
      RemoveMarks
      end
end
```

Figure 7 : Procédure Self (application de transitions asynchrones)

L'application d'une transition $s \xrightarrow{\tau} s' \in T_j$ à un noeud n ∈ Nj : val(n) = s s'effectue donc en deux étapes :

- Créer un nouveau noeud n' de valeur val(n') = s' dans la couche j de l'AP et lui ajouter tous les successeurs de n.
- Ajouter l'ensemble des éléments de l'ASP de racine n' comme successeur de tous les noeuds $f \in N_{j-1}$: n ∈ succ(f).

Le marquage des noeuds visités permet de n'effecuer qu'une seule fois l'opération de création d'un nouveau noeud pour chaque transition.

La procédure Self(j:integer) réalise donc l'opération Y = Y ∪ $Self_j(Y)$ vue dans le paragraphe précédent.

Grâce au marquage, chaque noeud des couches j et j-1 est traversé au plus une fois pour chaque transition asynchrone de l'automate Aut_j. L'algorithme a donc une complexité théorique de $O(\#N \times \#T_j)$.

```
procedure Synchro(in j:integer);
var
  k          : integer;
  s,sj',sk'  : state;
  n,newnode  : node;

  procedure RecSyncTrans(in n:TNode; in depth:integer)
  var n': TNode;
  begin
    n.marked := true;
    case depth of
      depth < j : % calcul les résultats pour les successeur
        forall n' ∈ n.succ do begin
          if not n'.marked
          then RecSyncTrans(n',depth+1);
          if n'.result <> null
          then AddGraph(n,n'.result)
        end
      depth = j : % calcul de la première partie de synchro
        if n.value = s
        then begin
          forall n' ∈ n.succ do
            if not n'.marked
            then RecSyncTrans(n',depth+1);
          if ∃n' ∈ n.succ : n'.result <> null
          then begin
            newnode := AddNode(depth,sj');
            forall n' ∈ n.succ do
              if n'.result <> null
              then AddSucc(newnode, n');
            n.result := newnode;
          end
        end
```

```
      depth = k : % calcul de la seconde partie de synchro
        forall n.value --ā--> sk' ∈ Tk do begin
          newnode := AddNode(depth,sk');
          forall n' ∈ n.succ do
            AddSucc(newnode,n');
          n.result := newnode
        end
      j < depth < k : % continue avec les successeurs
        forall n' ∈ n.succ do
          if not n'.marked
          then RecSyncTrans(n',depth+1);
        if ∃ n' ∈ n.succ : n'.result <> null
        then begin
          newnode := AddNode(depth,n.value);
          forall n' ∈ n.succ do
            if n'.result <> null
            then AddSucc(newnode,n');
          n.result := newnode
        end
      end;

begin
  forall s ∈ Sj do
    forall s --a--> sj' ∈ Tj do
      forall k : ā ∈ Āk do
        if k > j
        then begin
          forall n ∈ Nj-1 do
            RecSyncTrans(n,j-1);
          RemoveMarks
        end
end
```

Figure 8 : Procédure synchro (calcule les transitions synchrones d'un niveau j et k (k>j))

La Figure 9 montre schématiquement les opérations effectuées par synchro et il est intéressant de suivre le déroulement de cette procédure sur cet exemple. Ceci permet de mieux comprendre les opérations qui y sont effectuées. On ne considère ici que les chemins "gagnants" (c.-à-d. les chemins qui aboutissent à l'application d'une transition synchrone), les autres ne correspondant qu'à des renvois de *n.result = null* pour annuler l'opération.

Les chiffres sur le schéma indiquent l'ordre des opérations. Les "M" indiquent les noeuds marqués lors de l'entrée dans RecSyncTrans. Les noeuds grisés sont les deux noeuds à partir desquels il est possible d'effectuer une transition synchrone. Les pointillés indiquent la valeur du champ result pour les noeuds marqués. Enfin les éléments en gras sont ceux qui sont créés durant l'exécution de la procédure.

Dans l'algorithme synchro appelé avec le paramètre j, on va donc réaliser les transitions du type $s \xrightarrow{a} s_j' \in T_j$. La procédure réalise cela de la manière suivante : pour tous les automates

Aut_k comprenant la transition complémentaire à a $\bar{a} \in \overline{A}_k$ [1], si k > j on appelle la sous-procédure RecSyncTrans avec tous les noeuds du niveau inférieur. Il est en effet indispensable de pouvoir rattacher les résultats obtenus aux niveaux inférieurs.

Au premier appel de RecSyncTrans, on se trouve donc toujours dans le cas *depth* < *j*. On peut alors appeler RecSyncTrans récursivement jusqu'à ce que soit on trouve que la transition est infaisable pour le chemin suivi (*depth=j* & *n.value* <> *s* **ou** *depht=k* & il n'existe pas de noeud n tq $n.value \xrightarrow{\bar{a}} s_k' \in T_k$), soit on réalise la transition (*depth=k*) et on crée l'arbre résultat lors des retours d'appels.

La difficulté de cet algorithme réside dans le fait qu'il faut réaliser non plus une seule transition élémentaire sur un niveau donné, mais deux transitions élémentaires à des niveaux distincts. D'où la nécessité de parcourir les noeuds se trouvant sur tous les chemins reliant des noeuds de synchronisation possible de manière à pouvoir relier les noeuds résultats des transitions élémentaires impliquées aux niveaux supérieur et inférieur.

On peut remarquer, tout comme pour la procédure *self* que l'algorithme parcourt une seule fois chaque noeud dans les couches j et j-1, et ceci pour chaque transition synchrone déclarée dans l'automate Aut_j. Ceci implique que l'algorithme *synchro* a une complexité théorique $0(\#N \times \#T_j)$

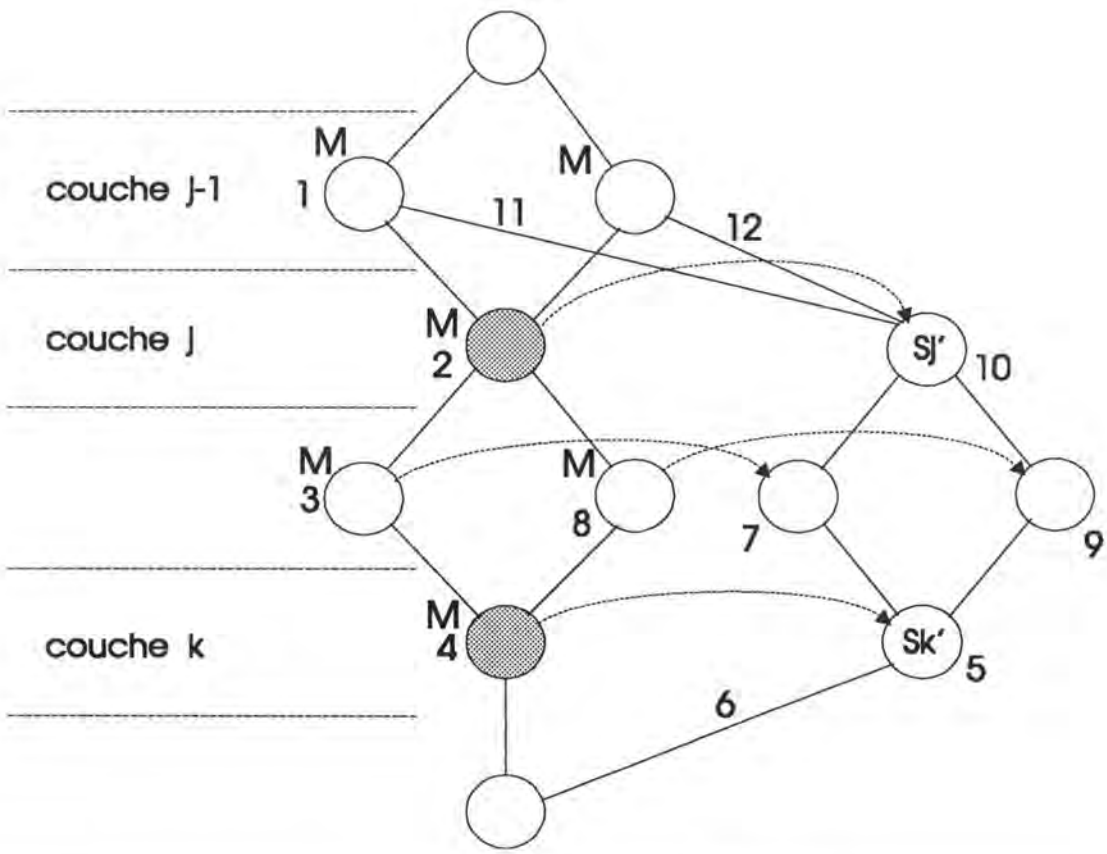

Figure 9 : Représentation graphique d'une exécution de synchro

[1] Dans notre formalisme, une transition synchrone détermine les deux automates impliqué par cette synchronisation. Il n'y a donc dans ce cas qu'un seul automate comprenant la transition complémentaire à a.

Les procédures *self* et *synchro* sont incluses dans une procédure générale qui itère en appelant successivement les procédures *self* et *synchro* pour chaque couche de l'automate produit jusqu'à ce que l'on trouve le point fixe, c'est-à-dire tant qu'au moins une modification est apportée à l'arbre partagé PSync.

2.2.3. Les optimisations apportées : le "caching" des opérations

Dans l'algorithme décrit précédemment, on peut remarquer qu'à chaque itération de la procédure principale, toutes les opérations effectuées précédemment (c.-à-d. les transitions sur des états déjà existants) sont refaites. En effet, l'algorithme de base ne permet pas de distinguer anciens noeuds non modifiés et nouveaux noeuds ou noeuds modifiés. Nous allons voir dans cette section comment il est possible d'optimiser l'algorithme en enregistrant les opérations déjà effectuées.

Tout d'abord, il est nécessaire de pouvoir déterminer quand il est superflu d'effectuer une transition à partir d'un noeud donné. C'est le cas uniquement si les deux conditions suivantes sont respectées :

- Le nombre de pères du noeud est inchangé depuis la dernière itération.
- Le nombre d'éléments de l'ASP de racine ce noeud est inchangé depuis la dernière itération.

La première condition asssure qu'aucun père n'a été ajouté au noeud. En effet, si on ajoute un père, toutes les transitions à partir du noeud donné doivent être exécutées une nouvelle fois de manière à ce qu'on puisse relier le résultat de celles-ci au nouveau père du noeud.

La seconde condition assure que toutes les transitions possibles à partir de ce noeud ont été effectuées lors des itérations précédentes. En effet, si l'ASP sous-jacent a été modifié, cela signifie qu'il est possible que certaines nouvelles transitions puissent être effectuées ou encore que le résultat d'autres transitions puisse être différent du précédent. Or le nombre d'éléments de tout sous-ASP de l'arbre PSync est croissant. En effet, on peut remarquer que les algorithmes analysés ci-dessus n'effectuent que des ajouts dans l'arbre. La condition de modification d'un sous-arbre revient donc à une augmentation du nombre d'éléments représentés par celui-ci.

Par conséquent, si les deux conditions sus-citées sont vérifiées, il est inutile de effectuer à nouveau les transitions à partir du noeud donné.

Ces considérations ont incité les auteurs de cet algorithme à introduire une optimisation consistant à stocker dans une table de hachage les opérations déjà effectuées sur les noeuds de l'arbre. Les différents éléments nécessaires aux tests des conditions 1 et 2 ont été joints à ces enregistrements. Ceci permet d'évaluer à tout moment la nécessité d'effectuer une transition donnée.

La table de hachage contiendra donc les informations suivantes :

- *ident* : l'identificateur du noeud de départ de l'opération.
- *nbFathers* : le nombre de pères du noeud au moment de l'enregistrement.
- *nbElem* : le nombre d'éléments de l'ASP de racine ce noeud au moment de l'enregistrement.
- *trans* : la transition effectuée après l'enregistrement.

On ajoute ensuite aux algorithmes de base des procédures permettant de savoir à tout moment le nombre d'éléments de l'ASP d'un noeud donné et le nombre de pères de ce noeud. Ceci implique évidemment une certaine perte de temps en gestion de ces informations, mais, comme nous le verrons dans la section suivante, les performances n'en souffrent aucunement. D'autre par, des procédures annexes ont été construites permettant l'insertion, la modification et la lecture d'enregistrement de la table.

Avant chaque transition, on teste donc si celle-ci doit réellement être effectuée. Si oui, on insère les éléments nécessaires à l'enregistrement de l'opération dans la table de hachage, si non (l'opération est déjà enregistrée), on passe à la transition suivante.

2.3. Résultats obtenus

L'algorithme a été implémenté dans un environnement PC compatible IBM sous WINDOWS. Les tests on été effectués avec un processeur Intel 486 DX2/66.

2.3.1. Un exemple typique : Les Schedulers de Milner

La Figure 10 nous montre l'exemple d'un automate représentant le $j^{ème}$ scheduler d'un système. Un scheduler est un automate qui attend un signal c_j venant du scheduler précédent avant de réaliser une action a_j. Ensuite, il envoie un signal c_{j+1} au scheduler suivant, soit avant, soit après avoir effectué une série d'opérations internes. Un automate appelé starter sert à lancer le système en envoyant un signal c_1 au scheduler n°1. Le dernier scheduler envoie son message au premier, ce qui permet au système de boucler indéfiniment.

Les schedulers sont des exemples pratiques pour l'évaluation des algorithmes de manipulation d'automates travaillant en parallèle. En effet, l'avantage des schedulers est que l'on peut construire des systèmes d'autant de schedulers que l'on veut. Ainsi, on peut tester simplement le comportement d'un algorithme face à des systèmes de tailles croissantes. De plus, l'ensemble des états du produit synchronisé de ces automates croît de manière exponentielle par rapport au nombre d'automates.

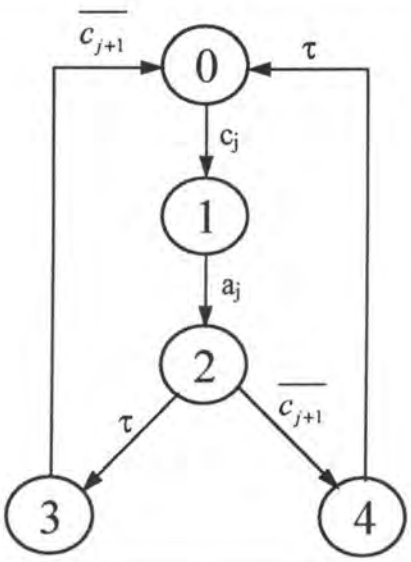

Figure 10 : Scheduler de Milner

2.3.2. Les résultats obtenus avec et sans optimisation

# auto.	6	8	10	12	14	16	18	20...	30
# états	577	3073	15361	73729	$34\ 10^4$	$15\ 10^5$	$7\ 10^6$	$31\ 10^6$	$48\ 10^9$
# noeuds	46	62	78	94	110	126	142	158	238
# fils	108	150	192	234	276	318	360	402	612
temps 1	0.28	0.99	6.59	85.3	-	-	-	-	-
temps 2	0.11	0.22	0.33	0.44	0.60	0.83	1.06	1.30	2.96

Temps 1 représente les temps d'exécution en secondes lorsqu'on n'utilise pas le caching.
Temps 2 représente les temps d'exécution en secondes lorsqu'on utilise le caching.

Tableau 1 : Résultats de l'algorithme DZA&BLE sur les Schedulers de Milner

Les résultats exposés dans le Tableau 1 nous amènent à faire les remarques suivantes :

Tout d'abord, on peut remarquer que, bien que le nombre d'états de l'automate produit croisse exponentiellement, le nombre de noeuds et d'arcs (nombre de fils) de l'AP représentant les états accessibles du système croît, lui, de manière linéaire. Ceci implique la possibilité de stockage de plusieurs milliards d'états en quelques centaines de bytes seulement. La structure d'Arbre Partagé montre dans ce cas pleinement ses capacités réelles de stockage.

De plus, le nombre d'itérations de l'algorithme pour calculer ces états, c'est-à-dire pour atteindre le point fixe de la transformation n'est pas non plus une fonction du nombre d'états recherchés mais bien du nombre de noeuds créés dans le graphe.

Cependant, on remarque que les temps de calcul de l'algorithme, lorsque l'on n'utilise pas de caching (optimisation sous forme de table de hachage), sont, eux, exponentiels. On peut expliquer cela par le fait que l'on effectue à chaque itération toutes les opérations possibles au lieu de calculer seulement les nouveaux états (création de nouveaux noeuds).

L'optimisation apportée permet de ramener l'algorithme à la notion intuitive que l'on en a, c'est-à-dire à la recherche à chaque itération de nouveaux éléments. Cette optimisation permet de réduire les temps de calcul jusqu'à les rendre linéaires.

On obtient ainsi un algorithme de calcul à la fois économe en espace mémoire et en temps de calcul.

3. Comparaison avec les algorithmes les plus connus

Certains algorithmes de calcul des états accessibles du produit synchronisé d'un système d'automates avaient déjà été mis au point avant celui de DZA&BLE. Ces premiers travaillaient sur d'autres structures de données (les BDDs par exemples) et utilisaient donc des moyens de recherches différents.

Les comparaisons effectuées entre ces différentes techniques sur l'exemple des schedulers (Tableau 2) montrent la supériorité point de vue performances de l'algorithme DZA&BLE sur tous les autres.

# automates	6	8	10	12	14	16	18	20
# états	577	3073	15361	73729	$34\ 10^4$	$15\ 10^5$	$7\ 10^6$	$31\ 10^6$
Fernandez[1]	2.6	21	160	-	-	-	-	-
Groote[2]	0.2	1.2	7.4	53	-	-	-	-
Enders[3]	21	40	87	145	233	348	569	850
Rauzy[4]	0.7	1.3	2.03	3.06	4.41	5.76	7.36	9.36
DZA&BLE[5]	0.11	0.22	0.33	0.44	0.60	0.83	1.06	1.30

* Tous les temps sont exprimés en secondes.
1. Aldébaran, IMAG Grenoble, temps mesurés sur SUN 3.
2. BB, CWI Amsterdam, SUN 3.
3. Système basé sur les BDDs, ZFE Munich, SUN 4.
4. Toupie, LABRI Bordeaux, SUN 4
5. Automata Analyser, Namur, PC 486 DX2/66

Tableau 2 : Comparaison de temps d'exécution [1]

La technique de Enders est la plus proche de celle utilisée par DZA&BLE. La structure de donnée utilisée est le BDD (Binary Decision Diagram). L'inconvénient majeur de cette structure est le suivant : malgré une place de stockage très réduite, la manipulation de ces données requiert encodage et décodage des données initiales. De plus, les BDDs intermédiaire

qui peuvent être générés durant le calcul sont grands et nécessitent donc plus de mémoire disponible.

4. Conclusions et alternatives

L'algorithme DZA&BLE, testé sur la machine la plus lente, reste de loin le plus rapide (cfr. Tableau 2). Ceci nous fait supposer que la structure des arbres partagés est la plus adéquate qui ait été expérimentée.

Dans ce mémoire, nous avons décidé d'explorer la voie d'un autre algorithme basé sur les arbres partagés. En effet, l'algorithme proposé dans ce chapitre n'est pas le seul possible. Nous allons voir qu'il y a moyen de considérer la recherche de l'ensemble des états accessibles non plus comme une recherche de point fixe global, mais comme un développement successif de branches partielles de l'arbre partagé.

Chapitre 3
Un nouvel algorithme de calcul d'accessibilité

1. Introduction

Le chapitre précédent nous a montré un algorithme de calcul des états accessibles d'un produit synchronisé travaillant sur la structure des APs. Nous présentons dans ce chapitre un nouvel algorithme que nous avons implémenté, utilisant la même structure que l'algorithme DZA&BLE, mais ayant une technique de développement des états tout-à-fait différente.

Le chapitre s'organisera comme suit :

Premièrement nous exposerons l'idée générale de l'algorithme. Ceci nous permettra d'emblée une comparaison des différentes techniques. Ensuite, nous analyserons l'algorithme de manière approfondie et sur le plan théorique. Cette première partie nous donnera une structure initiale de l'algorithme sur laquelle sera basée l'implémentation.

Dans la deuxième partie, nous analyserons en détails l'implémentation de l'algorithme sur la même base que l'implémentation de DZA&BLE. Cette implémentation nous donnera des résultats à comparer avec ceux de DZA&BLE avant toute optimisation.

La troisième partie sera consacrée aux différentes optimisations apportées à l'algorithme. Nous verrons les différents moyens utilisés, leurs implémentations ainsi que les résultats obtenus.

Nous finirons le chapitre par une brève analyse des résultats obtenus et une critique de l'algorithme a posteriori.

2. Idée générale de l'algorithme

L'algorithme que nous avons implémenté, bien que travaillant sur les mêmes structures, est assez différent de celui de DZA&BLE. Bien que ces deux algorithmes recherchent l'ensemble des états accessibles d'un produit synchronisé, la technique utilisée est tout à fait différente.

Dans notre algorithme, on n'utilise pas la définition de l'ensemble des états accessibles comme le point fixe d'une transformation. On se sert en fait d'une liste qui définit à un niveau donné, l'ensemble des états à partir desquels on peut espérer effectuer au moins une "nouvelle" transition. Cette liste est mise à jour chaque fois que l'on effectue une transition de manière à ce que les nouveaux états accédés soient insérés dans cette liste. Le critère d'arrêt ou de passage au niveau précédent sera toujours le fait que cette liste soit vide. Le critère de terminaison de l'algorithme est d'avoir développé entièrement le niveau le plus haut.

En effet, il faut se rappeler que dans la structure d'AP développée, chaque niveau de l'AP correspond à un automate du système. L'algorithme effectue donc à un niveau i, toutes les transitions possibles de l'automate Aut_i, étant donné les états déjà accédés et se trouvant à ce niveau. Lorsqu'on a développé le niveau 1, cela signifie que toutes les transitions possibles au niveau 1 (pour l'automate Aut_1) ont été effectuées et tous les niveaux inférieurs ont été entièrement développés. Nous avons donc ainsi atteint l'ensemble des états accessibles du système.

La deuxième différence majeure est que l'on travaille non pas avec l'arbre entier comme le fait DZA&BLE, mais plutôt avec des sous-arbres (des branches d'arbres) que l'on développe successivement.

De plus, notre algorithme développe le système du bas vers le haut, c'est-à-dire que l'on construit les différentes branches en partant des feuilles et non pas de la racine.

Nous verrons dans le cours du chapitre les avantages et inconvénients de ces différences et en quoi elles contribuent à une amélioration ou à une déterioration des performances.

3. Définitions préliminaires

3.1. Les projections

Soit un ensemble E de nuplets. Alors on appellera projection de l'ensemble E sur sa $i^{ème}$ composante l'ensemble

$$proj_i(E) = \{s_i \; t.q. \; \exists (s_1, \ldots, s_{i-1}, s_i, s_{i+1} \ldots, s_n) \in E\}$$

De même, on définit la projection complémentaire à $proj_i(E)$ de la manière suivante :

$$\overline{proj_i(E)} = \{(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) \; t.q. \; \exists \, s_i \in proj_i(E) \wedge (s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n) \in E\}$$

Enfin, on définit la projection de l'ensemble E sur les composantes i à j avec 1 <= i <= j <= n l'ensemble

$$proj_{(i,\ldots,j)}(E) = \{(s_i, \ldots, s_j) \; t.q. \; \exists (t_1, \ldots, t_n) \in E \wedge (t_i, \ldots, t_j) = (s_i, \ldots, s_j)\}$$

3.2. Concepts d'automates

Soit un système d'automates $Aut_j = < S_j, i_j, T_j >$ (1 <= j <= n).

On dit que l'automate **Aut_i** est de niveau **plus bas** (resp. plus haut) que l'automate **Aut_k** si et seulement si **i > k** (resp. i < k).

Considérons l'ensemble des états accessibles du produit synchronisé de ce système d'automate (noté PSync).

Alors un état partiel de Psync (de niveau k) est un élément de l'ensemble $proj_{<k,...,n>}$(PSync). Un état partiel est donc la projection d'un état accessible du produit synchronisé sur ses dernières composantes.

Soit $t_j \in T_j$ et $s_j \in S_j$. Alors on définit le successeur de s_j pour la transition t_j de la manière suivante :

$$succ_{t_j}(s_j) = s'_j \quad \Leftrightarrow \quad s_j \xrightarrow{t_j} s'_j$$

Soit PSync' un sous-ensemble de PSync. Alors on parlera de transition utile sur PSync' quand cette transition permet de trouver le nouvel état du produit syncrhonisé (c.-à-d. quand l'application de cette transition à chaque état auquel elle peut s'appliquer dans PSync' produit au moins un état n'appartenant pas à PSync'). Les autres transitions seront appelées transitions inutiles pour l'ensemble PSync'.

4. Analyse de l'algorithme de base

Dans cette section, nous allons exposer l'algorithme de manière théorique. Il n'est cependant pas possible de faire abstraction de la structure de données pour laquelle il a été conçu. C'est pourquoi on verra parfois apparaître le mot "noeud" au lieu d' "états" sans toutefois faire mention explicitement d'arbre partagé. Les projections d'ensembles que l'on rencontre dans ce paragraphe correspondent elles-mêmes à des morceaux d'arbre partagé. Nous pensons cependant qu'il est préférable pour une bonne compréhension d'examiner d'abord l'algorithme sans chercher absolument à percevoir la structure sous-jacente.

4.1. Enoncé du problème

Soit un système d'automates $Aut_j = < S_j, i_j, T_j >$ ($1 <= j <= n$). Construire l'ensemble E des états accessibles du produit synchronisé de ce système d'automates.

4.2. Structure initiale de l'algorithme

L'algorithme peut être décomposé en trois procédures principales.

La procédure **Propagate** permet d'effectuer l'ensemble des transitions possibles à partir d'un ensemble d'états donné.

La procédure **Transmit** sert à effectuer une transition synchrone, et à développer tous les nouveaux noeuds créés à cette occasion.

La procédure **ConstSync** permet de construire l'ensemble des états accessibles en utilisant les deux premières procédures.

Ces trois procédures sont très liées puisque ConstSync appelle Propagate qui appelle Transmit. Transmit est une procédure récursive et appelle elle-même Propagate. Il est donc impossible de construire l'une sans évoquer les deux autres.

4.2.1. La procédure Propagate

La procédure Propagate travaille au niveau d'un automate Aut_{niv} donné. Elle développe l'ensemble des états de Aut_{niv} contenus dans une liste d'entrée jusqu'à ce qu'aucune opération ne puisse être effectuée avec les niveaux inférieurs (au sens donné dans 3.2).

Les paramètres d'entrée de Propagate sont donc les suivants :

- *E* : Ensemble de tuples de la forme $(s_{niv}, s_{niv+1}, \ldots, s_n)$.
  E est l'ensemble de base du développement. Il représente un sous-ensembles des états partiels de niveau *niv* (cfr. 3.2) déjà rencontrés.
- *niv* : Entier.
  Numéro de l'automate auquel les états de *slist* appartiennent.
- *slist* : Liste d'états.
  slist représente la liste des états à développer de l'automate Aut_{niv}. Cette liste est exhaustive, c'est-à-dire qu'il n'est pas possible d'effectuer une transition partant d'un état de $proj_1(E)$ n'appartenant pas à cette liste.
- *adevelop* : Tableau des états -> Bool.
  Pour tout état $s \in Aut_{niv}$, *adevelop*(s) = true ssi $s \in$ *slist* (optimisation).

Les paramètres de sortie de Propagate sont :

- *E'* ~ *E* final : Ensemble de tuples
  Ensemble des tuples accessibles à partir de E en appliquant l'ensemble des suites de transitions asynchrones ou synchrones avec les niveaux inférieurs (suites qui sont issues des états de *slist*).

  De manière plus formelle, on peut définir E' de la manière suivante :

  Soit T l'ensemble des transitions possibles du produit synchronisé PSync.
  On restreint cet ensemble T en ne considérant que les transitions impliquants des automates de niveau i >= *niv*. Soit T' cet ensemble.
  Soient $S' = S_{niv} \times \ldots \times S_n$ et $I' = (i_{niv}, \ldots, i_n)$.
  Alors $s \in E'$ si et seulement s'il existe une suite de transitions $(t_i)_{i \in [1..m]} \subseteq T'$ et une suite d'états $(s_i)_{i \in [0..m]} \subseteq S'$ telles que :

  $$s_0 \in E$$
  $$s_m = s$$
  $$t_i = (s_{i-1}, a_i, s_i) \ \forall\, i = 1..m$$

La Figure 11 présente l'algorithme de Propagate.

```
Procedure Propagate(slist:List of Etats;adevelop:Etats->Bool;
                    var E:Set of tuples;niv:integer);

var
  s,s' : State;
  E',E'' : Set of tuples;
  sync : integer;

begin
  while slist <> [ ] do
    begin
      % Prendre le premier élément de la liste slist
      s := head(slist);
      slist := tail(slist);
      adevelop(s) := false;
      % Rassembler l'ensemble des états du produit comprenant s
      E' := {(s_{i+1}, ..., s_{n+1}):(s,s_{i+1}, ...,s_{n+1})∈E}
      % Boucle sur toutes les transitions asynchrones partant de s
      forall s --τ--> s' ∈ Aut_niv do
        begin
          % Détection des nouveaux états
          if (not adevelop(s') and (s'X E' ⊄ E) then
            begin
              adevelop(s') := true;
              slist := insert(s',slist);
            end;
          % Mise à jour de l'ensemble résultat
          E := E U (s' X E');
        end;
      % Boucle sur toutes les transitions synchrones
      forall s --a--> s' ∈ Aut_niv do
        begin
          % Recherche du niveau synchrone
          sync := Sync_niv(s --a--> s' ,niv);
          if sync > niv then
            begin
              % Appel de transmit pour effectuer l'autre transition
                élémentaire et les développements qui en découlent
              transmit(niv,sync,ā ,E',E'');
              % détection des nouveaux noeuds
              if (not adevelop(s') and (s'X E'' ⊄ E) then
                begin
                  adevelop(s') := true;
                  slist := insert(s',slist);
                end;
              % Mise à jour de l'ensemble résultat
              E := E U s' X E'';
            end;
        end;
    end;
end
```

Figure 11 : Algorithme de la procédure Propagate (point de vue théorique)

On remarque que le traitement des transitions asynchrones se fait facilement puisqu'il s'agit d'une transition élémentaire qui n'implique donc qu'un seul niveau. Par contre, le traitement des transitions synchrones, nécessitant deux niveaux, est plus complexe et doit être effectué par la procédure *Transmit* que nous présentons dans la section suivante.

Nous avons vu dans le paragraphe 4.2 que notre algorithme travaillait non pas sur l'arbre tout entier mais bien sur les branches de cet arbre. On peut déjà en trouver un indice sur cet algorithme théorique. En effet, on voit que l'algorithme travaille comme si le niveau courant *niv* était le premier niveau. A aucun moment on ne tient compte des niveaux supérieurs. De plus, chaque fois que l'on prend un élément *s* dans *slist*, on isole l'ensemble des tuples dont la première composante est *s*. Dans la suite, on ne considère plus que cet ensemble. C'est pourquoi on peut dire que le développement se fait par branche (ici la base (le point d'ancrage) de la branche est s).

A chaque passage par le *while*, *slist* représente la liste exhaustive des états à développer. En effet, au départ, c'est une condition initiale de la procédure. Ensuite, les critères d'insertion dans la liste *slist* (qui détermine les noeuds à développer) sont les suivants. Lorsqu'une transition est possible entre des noeuds *s* et *s'*, on examine si, après avoir effectué cette transition, des éléments supplémentaires ont été ajoutés à l'ensemble *E*. Si c'est le cas, alors *s'* doit être développé. Sinon, soit *s'* est déjà dans la liste, soit il ne faut pas l'y ajouter. En effet, le fait qu'il ne soit pas dans la liste implique qu'il n'est pas possible d'effectuer une transtion utile à partir de cet état et, comme la transition effectuée n'a pas modifié la situation, il n'est toujours pas possible de le faire (d'où aucune raison pour insérer cet état dans la liste).

L'invariant que nous avons détecté dans les lignes précédentes nous permet d'assurer que le E sera complètement développé lorsqu'on sortira de la procédure. En effet, la condition de terminaison est que la liste *slist* soit vide lorsqu'on passe dans le *while*, ce qui signifie qu'aucun état ne peut plus être développé et donc que l'ensemble E a été complètement développé.

4.2.2. La procédure Transmit

La procédure Transmit permet d'effectuer les transitions synchrones. Pour ce faire, elle descend depuis le niveau source vers le niveau synchrone où elle cherche à effectuer la transition complémentaire. En remontant dans les appels récursifs, on développe les noeuds créés ou modifiés. Ceci permet d'assurer qu'à tout moment, les niveaux plus bas que le niveau courant sont entièrement développés.

Les paramètres nécessaires au bon fonctionnement de Transmit sont les suivants :

Paramètres d'entrée :

- *source*, *sync* : Entiers.
  Numéros d'automates tels que $1 \leq source < sync \leq n$.
- *t* : Transition.
  Transition élémentaire à effectuer au niveau *sync*.
- *E'* : Ensemble de tuples d'états de la forme $(e_k, ..., e_{n+1})$ avec k fixé tel que $i < k \leq j$.
  Ensemble de base du développement effectué. Cet ensemble est entièrement développé (seule la transition élémentaire *t* peut provoquer de nouveaux développements).

Paramètre de sortie :

- E'' : Ensemble de tous les tuples.
  E'' représente l'ensemble des tuples de la même forme que ceux de E' accessibles à partir de tuples de E' en faisant toutes les transitions possibles dans les automates Aut_k, ..., Aut_n, avec la possibilité de communication entre ces automates et en commençant par une transition synchrone t avec des états synchronisés au niveaux j et i.

  Plus formellement,

  Soit T' la restriction de T à l'ensemble des transitions impliquant des automates de niveaux i >= niv. Soient $S' = S_{niv} \times ... \times S_n$ et $I' = (i_{niv}, ..., i_n)$.
  Alors s ∈ S' appartient à l'ensemble E'' si et seulement s'il existe une suite de transition $(t_i)_{i\in[1..m]} \subseteq T'$ et une suite d'états $(s_i)_{i\in[0..m]} \subseteq S'$ telles que :

  $s_0 \in E'$
  $s_m = s$
  $t_1 = t$
  $t_i = (s_{i-1}, a_i, s_i)\ \forall\, i = 1..m$

L'algorithme de cette procédure est décrit à la Figure 12.

```
Procedure Transmit(source,sync:integer;
                   t : Transition; E':Set of tuples;
                   var E'':Set of tuples;
var
  s : state;
  S,S' : Set of State;
  Es,E'',E3 : Set of tuples;
  slist : list of State;

begin
  % Si on se trouve au niveau de la synchronisation
  % k est l'indice correspondant au premiers éléments des tuples de E'
  if k = sync then
    begin
      % S et (Es)s∈S sont t.q. E' = ⋃s∈S ({s} × Es)
      S := Proj1(E');
      forall s ∈ S do
        Es := {(xk+1,...,xn) : (s,xk+1,...,xn) ∈ E' };
      % Choisir l'ensemble des s où il y a transition possible
      S' := { s : s ∈ S et ∃ s --t--> s' ∈ Aut_sync }
      % Construire l'ensemble résultat E''
      E'' := ⋃s∈S' (succt(s) × Es) ;
      % Ajouter à slist l'ensemble des états du niveau sync à développer
      slist := Succt(S');
      % Mettre à jour la fonction adevelop
      for s ∈ State_sync - Succt(S') do adevelop(s) := false;
      for s ∈ Succt(S') do adevelop(s) := true;
      % Développer les états de slist grâce à Propagate
```

```
      Propagate(slist,adevelop,E'',sync);
    end
  else
    begin
      % S et (E_s)_{s∈S} sont t.q. E' = ⋃_{s∈S} ({s} × E_s)
      S := Proj_1(E');
      forall s ∈ S do
        E_s := {(x_{k+1},...,x_n) : (s,x_{k+1},...,x_n) ∈ E' };
      % Initialisation de E'' et de slist
      E'' := ∅;
      slist := [ ];
      forall s ∈ state_k do
        adevelop(s) := false;
      % Pour chaque état s du niveau k,
      % on appelle transmit avec l'ensemble E_s correspondant
      forall s ∈ S do
        begin
          transmit(source,sync,t,E_s,E^3);
          % Mise à jour de l'ensemble résultat
          E'' := E'' ∪ {s} x E^3;
          % Si des nouveaux noeuds ont été créés, les ajouter à la liste
          % des noeuds à développer et mettre à jour adevelop
          if E^3 <> ∅ then
            begin
              slist := insert(s,slist);
              adevelop(s) := true;
            end;
        end;
      % Développement des noeuds de slist grâce à Propagate
      Propagate(slist,adevelop,E'',k);
    end;
end;
```

Figure 12 : Procédure Transmit (le point de vue théorique)

Nous allons montrer que E'' représente bien l'ensemble des états accessibles à partir de E' en commençant par effectuer une transition élémentaire t au niveau sync.

Au niveau k = *sync*, il suffit de montrer qu'on développe complétement le niveau sync en commençant par la transition élémentaire *t* (et qu'il est attaché aux niveaux inférieurs) .

A ce niveau, le premier champ de chaque tuple de E' (sous-état de niveau j) est un état de Aut_j. On peut alors déterminer les états de niveau j pour lesquels la synchronisation est possible et on peut ainsi effectuer la transition élémentaire correspondante.

Si on admet que S' représente l'ensemble des états à partir desquels on peut effectuer la transition t, on voit que l'ensemble E'' construit par l'instruction $E'' := \bigcup_{s \in S'} (succ_t(s) \times E_s)$ représente l'ensemble des tuples accessibles à partir de E' en effectuant la transition *t*. Comme on a construit slist avec $Proj_1(E'')$, slist contient bien l'ensemble des états à développer du niveau sync de E''. En conséquence, comme nous l'avons vu dans le paragraphe précédent, le résultat de l'appel de Propagate est le développement complet du niveau sync de E''. Ceci entraîne que l'ensemble E'' construit par l'algorithme est bien celui spécifié.

Au niveau k < *sync*, qu'en est-il ?

Nous allons montrer par récurrence que, si la procédure Transmit est correcte au niveau k+1, elle est correcte au niveau k.

Pour chaque s $\in$ $Proj_1(E')$, on appelle Transmit avec E_s. Le résultat (par hypothèse de récurrence) est E^3, ensemble des noeuds accessibles à partir de E_s en effectuant toutes les suites de transitions possibles commençant par la transition élémentaire t au niveau sync. L'ensemble *E''* construit sera donc l'union de tous les ensembles de la forme {s} x E^3 ainsi construits. Comme par hypothèse de récurrence, tous les niveaux inférieurs ont été complètement développés, l'appel de Propagate avec *slist* = {s $\in$ S} et *E''* va permettre de développer entièrement le niveau k en faisant toutes les transitions possibles avec les niveaux inférieurs. En conséquence, l'ensemble *E''* ainsi construit est bien l'ensemble recherché.

Notons que ces considérations ne sont valables que si la procédure Propagate est correcte. Or, dans la procédure Propagate, on suppose Transmit correcte. Nous n'avons donc pas encore une preuve formelle de la correction de nos deux procédures. Pour ce faire, nous devons considérer les procédures Propagate et Transmit simultanément. Nous allons montrer par récurrence sur les niveaux que les procédures Propagate et Transmit sont correctes.

Tout d'abord, on peut dire que si les procédure Propagate et Transmit sont correctes pour les niveaux i > niv, alors la procédure Transmit est correcte pour le niveau niv. En effet, Transmit n'utilise Propagate que pour développer le niveau inférieur au niveau courant.

De même, la procédure Propagate est correcte au niveau niv si la procédure Transmit est correcte à ce même niveau.

Montrons premièrement que la procédure Propagate est correcte à l'avant-dernier niveau (niveau de l'automate Aut_n puisque le dernier niveau est celui de l'automate fictif). La procédure Transmit n'est jamais appelée à ce niveau (il n'y a pas de transition synchrone avec un niveau inférieur puisque c'est le dernier niveau non fictif) et donc implicitement est correcte pour ce niveau. Donc, la procédure Propagate est correcte elle aussi pour ce niveau.

Si les procédures Propagate Transmit sont correctes au niveau i, la procédure Transmit est correcte au niveau i-1. De plus, si la procédure Transmit est correcte au niveau i-1, Propagate l'est aussi.

Nous avons ainsi montré la correction des procédures Propagate et Transmit à tous les niveaux auxquels elles s'appliquent.

4.2.3. La procédure ConstSync

Les procédures Propagate et Transmit permettent de développer un niveau donné. Nous allons maintenant voir comment on peut utiliser ces deux procédures pour développer l'ensemble des états accessibles du produit synchronisé d'un système de n automates.

La procédure ConstSync permet de construire cet ensemble. Pour ce faire, on procède comme indiqué par la Figure 13.

```
Procedure ConstSync(var E : Set of tuples);

var
  slist : List of State;
  niv : integer;
  s : State;

begin
  % Mettre dans E l'état initial de l'automate produit
  E := {(i_n)};
  % Développer les états initiaux en partant du bas
  for niv := n downto 1 do
    begin
      slist := [i_niv];
      forall s ∈ States_niv do
        adevelop(s) := false;
      adevelop(i_niv) := true;
      Propagate(slist,adevelop,E,niv);
      if niv <> 1 then
        E := {i_niv-1} X E;
    end;
end;
```

Figure 13 : Algorithme de la procédure ConstSync (Point de vue théorique).

De nouveau, on peut remarquer que le développement se fait de bas en haut. Ceci permet d'assurer qu'à tout moment, les niveaux inférieurs au niveau courant ont été entièrement développés. On ne redescend dans les niveaux que lorsque l'on effectue une transition synchrone. Dans ce cas, comme nous l'avons vu précédemment, on effectue un redéveloppement de la partie de E modifiée.

4.3. Agencement des différentes procédures et fonctionnement global de l'algorithme

L'agencement des procédures est assez évident. Il s'agit donc d'appeler la procédure ConstSync qui utilise Propagate pour développer chaque niveau. La procédure Propagate utilise elle-même la procédure Transmit récursive. De même, Transmit utilise Propagate pour le développement des nouveaux états qu'elle crée.

Il nous reste à vérifier que les conditions initiales de chaque procédure sont vérifiées lors des appels.

4.3.1. Vérification des conditions initiales lors des appels aux procédures

Appel de Propagate dans ConstSync :

Il suffit de vérifier que slist est bien la liste exhaustive des états à développer. C'est forcémment vrai puisque slist contient l'ensemble des états du niveau en question.

Appel de Transmit dans Propagate :

Il faut vérifier que E' est bien complètement développé. Or nous avons vu précédemment que tous les niveaux inférieurs au niveau courant étaient entièrement développés. Donc la condition est bien vérifiée.

La procédure *transmit* permet de parcourir tous les chemins joignant l'état s_j de l'automate Aut_j aux états s_k de l'automate Aut_k ($j < k$) (états synchrones pour une transition t). Pour ce faire, on procède de la manière suivante :

On appelle transmit avec $k = source+1$. Le but étant de parcourir tous les chemins possibles, on va considérer à ce niveau tous les états de niveau k inclus dans les tuples de E'. Ceci permet de créer une partition de E' selon la première composantes. Chaque E_s ainsi créé va permettre d'appeler *transmit* récursivement comme si la synchronisation se faisait entre les noeuds s du niveau k et le niveau j.

Lorsqu'on sort de Transmit dans la procédure Propagate, le paramètre E" contient l'ensemble des sous-états de niveau source+1 accessibles à partir de E' en effectuant toutes les transitions possibles asynchrones et synchrones entre les niveaux de E'.

Appel de Transmit dans Transmit :

Idem Appel de Transmit dans Propagate.

Appel de Propagate dans Transmit :

slist contient bien la liste exhaustive des états à développer. En effet tous les états créés ou atteints lors de l'exécution de Transmit ont été insérés dans slist. Les autres états n'ont pas été atteints et donc pas modifiés, il ne doivent donc pas être insérés dans slist.

4.3.2. Un point de vue global de l'algorithme

Nous avons vu tout au long de l'analyse que l'algorithme développait les différents niveaux toujours du bas vers le haut. Cette technique permet d'assurer qu'à tout moment, les niveaux inférieurs au niveau courant sont entièrement développés. Cette propriété est très importante pour le bon fonctionnement de l'algorithme. En effet, elle est utilisée implicitement chaque fois que l'on insère des nouveaux sous-états à un niveau donné en effectuant l'opération

$$E' := E' \cup \{s\} \times E''$$

Il est indispensable de pouvoir assurer que tous les développements ont été effectués dans E". Sinon, certains états accessibles ne seraient jamais atteints par notre algorithme. C'est pourquoi toutes les procédures développent chaque fois complètement un niveau avant de remonter au niveau supérieur.

On peut d'ores et déjà remarquer les points cruciaux de l'algorithme. D'une part, on peut voir dans Propagate des instructions difficiles telles que le test de l'inclusion d'un ensemble dans un autre ou encore l'union de deux ensembles. Nous verrons dans les sections suivantes comment

ces instructions a priori lourdes vont être implémentées de manière ne pas entraver les performances de l'algorithme.

D'autre part, il n'est pas possible d'exprimer notre algorithme sous la forme d'itérations comme c'était le cas dans l'algorithme DZA&BLE. Il n'y a pas de recherche de point fixe à proprement parler. On voit d'un côté que l'algorithme utilise une technique de stabilisation par niveau. En effet, la procédure Propagate a pour but le développement d'un niveau entier lorsqu'elle est appelée par ConstSync. Par ailleurs, cette stabilisation se fait souvent sur une partie de niveau seulement. C'est le cas lorsque la procédure Propagate est appelée à partir de la procédure Transmit. C'est pourquoi on parlera de notre algorithme comme d'une technique de stabilisation partielle par niveau.

Un avantage que semble offrir l'algorithme, a priori, est de ne pas effectuer plusieurs fois les mêmes opérations. En effet, que ce soit dans Propagate ou dans Transmit, on remarque qu'un état n'est ajouté à slist que s'il a été la destination d'une transition qui a modifié l'ensemble des états accessibles, ce qui signifie qu'il peut être l'origine de nouvelles transitions.

5. Implémentation de l'algorithme de base

5.1. Introduction

Dans cette partie, nous allons traduire l'algorithme que nous venons d'analyser en terme de structure PASCAL et d'arbre partagé. Pour ce faire, nous présenterons après une série de définitions, l'environnement dans lequel nous avons implémenté l'algorithme, les structures de données utilisées, et nous analyserons l'implémentation des différentes procédures en les comparant avec les procédures théoriques.

5.2. Définitions préliminaires

Nous ne reviendrons pas sur la définition d'arbre partagé (AP) et d'arbre sous-partagé (ASP) vue dans la partie théorique. Cependant, dans la suite, il est nécessaire de définir d'autres concepts ayant rapport avec cette structure de données.

On définit ainsi :

Un sous-AP (resp. sous-ASP) de racine r (où r est un noeud de niveau j) est la partie de l'AP (resp. ASP) correspondant à la projection de l'ensemble des éléments de l'AP (resp. ASP) comprenant r sur ses n-j+1 dernières composantes. La Figure 14 permet de se rendre compte de la réalité physique des sous-AP (resp. sous-ASP). On y a encadré en traits discontinus gras le sous-AP de racine r (noeud de valeur 1 sur la couche 2).

L'ensemble des éléments d'un sous-AP (resp. sous-ASP) de racine r est donc l'ensemble des tuples se trouvant sur les chemins partant de la racine r du sous-AP (resp. sous-ASP).

On définit également les descendants d'un noeud n d'un AP (resp. ASP) par l'ensemble des éléments représentés par les sous-AP (resp. ASP) de racine les fils de n. Ainsi, l'ensemble des descendants du noeud r à la Figure 14 (le noeud en gras) est l'ensemble

$$E := \{(5,1), (5,6)\}$$

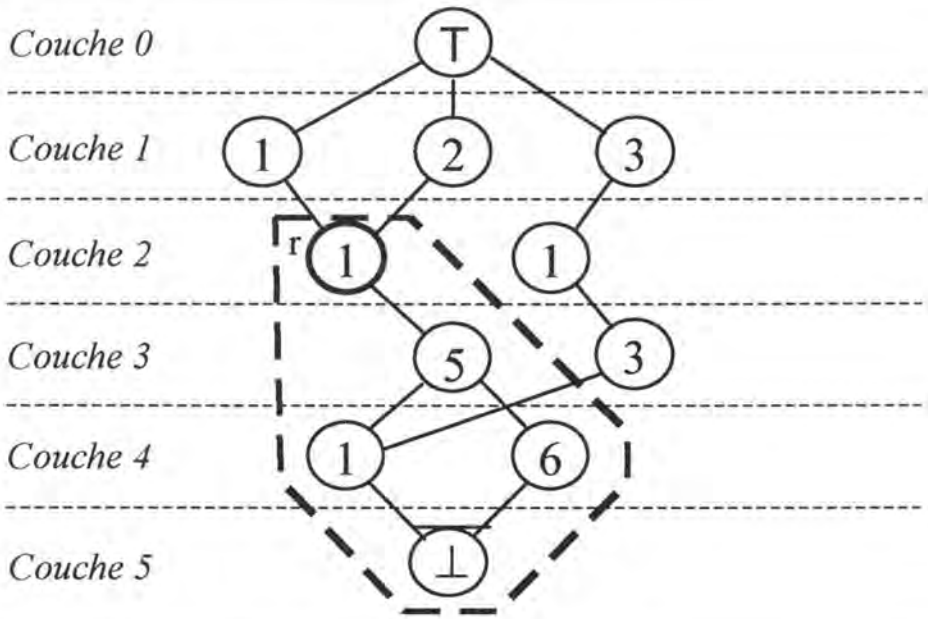

Figure 14 : Représentation graphique d'un sous-AP.

5.3. Présentation de l'environnement dans lequel l'algorithme a été implémenté

5.3.1. Environnement matériel

L'algorithme a été implémenté sur une machine de type PC compatible IBM dans l'environnement WINDOWS. Cet environnement a été imposé par le fait que l'on allait travailler sur les mêmes structures de données et avec la même interface que l'algorithme DZA&BLE.

5.3.2. Environnement logiciel

Nous avons implémenté l'algorithme sur la même structure que celui de DZA&BLE. Cet environnement offre les possibilités suivantes :

- Un langage simple et intuitif de définition des automates que l'algorithme utilise pour sa recherche (voir Figure 15).

- Les structures de données nécessaires au stockage des descriptions des automates (voir 5.4.1)
- Des procédures d'interprétation du langage de définition permettant le transfert vers les structures de données.
- Une interface convivial sous WINDOWS (que nous avons modifié pour y intégrer notre algorithme.

De plus, cet environnement inclut l'ensemble des procédures nécessaires à la manipulation des Arbres Partagés ainsi que l'algorithme de DZA&BLE. Il est ainsi possible d'effectuer des comparaisons rapides des performances des deux algorithmes.

```
% Milner's schedulers
% second scheduler process

% Ensemble des états de l'automate
s0,s1,s2,s3,s4

% Ensemble des transitions de l'automate
% Transition synchrone avec p0 de nom ~start1 (complémentaire de start1)
% allant de l'état s0 à s1
p0: ~start1 = s0 > s1;
% Transition asynchrone de nom a1 allant de l'état s1 à s2
self: a1 = s1 > s2;
self: choice = s2 > s4;
p2: start2 = s2 > s3;
self: back = s3 > s0;
p2: start2 = s4 > s0

end
```

Figure 15 : Description d'un scheduler dans le langage de DZA&BLE

5.4. Les structures de données utilisées

5.4.1. La structure de stockage des automates

Comme nous l'avons entrevu dans le paragraphe précédent, l'environnement dans lequel nous avons choisi d'implémenter l'algorithme comprenait une structure de données pour le stockage des automates.

Un automate sera défini par un nom, un ensemble d'état et un ensemble de transitions. Chaque état donne accès à toutes les transitions ayant cet état pour source et chaque transition donne accès aux deux états impliqués. En outre, on associe aux transitions synchrones un champ *synchronisateur* qui permet de savoir avec quel autre automate la transition est synchrone et un champ booléen *initiateur* pour déterminer s'il s'agit de la partie "émettrice" ou "réceptrice" de la transtion (t ou $\bar{t}$). La Figure 16 nous donne une description en PASCAL de cette structure de données.

Un système d'automates est stocké sous la forme d'un tableau de PAutomates appelé *Automates*, ce qui permet d'associer à chaque automate du système traité un numéro (son niveau) qui est l'entrée de cet automate dans le tableau Automates.

```
string20 = string[20];
Pautomate = ^Tautomate;
Ptransition = ^Ttransition;
Petat = ^Tetat;

{ un etat d'un automate est defini par un nom, et une liste de
  (pointeurs vers) transitions possibles a partir de cet etat }
Tetat = record
        nom: string20;
        SelfTransPoss: array[1..MaxTransPoss] of Ptransition;
        SyncTransPoss: array[1..MaxTransPoss] of Ptransition;
        nbSelfTransPoss,nbSyncTransPoss: integer
        end;

{ une transition entre etats d'un automate est definie par un nom,
  un (pointeur vers) automate synchronisateur de cette transition
  (0 = self), un flag indiquant si l'automate est initiateur de la
  transition, et les (pointeurs vers) etats origine et destination }
Ttransition = record
              nom: integer;
              synchronisateur: integer;
              initiateur: boolean;
              origine, destination: integer;
              next: Ptransition
              end;

{ un automate est defini par un nom, un ensemble d'etats, et un
  ensemble de (pointeurs vers) transitions entre ces etats }
Tautomate = record
            nom: string20;
            etat: array[1..MaxEtats] of Petat;
            nbEtats: integer;
            Ltransitions: Ptransition;
            end;
```

Figure 16 : Structure de données pour le stockage de la description d'automates

5.4.2. La structure de données pour le stockage des résultats

Nous avons utilisé pour le stockage des états accessibles la même structure de donnée que DZA&BLE, en y incluant des champs supplémentaires permettant le stockage d'informations supplémentaires nécessaires au bon fonctionnement de l'algorithme.

La définition de la structure de données d'un arbre partagé est donnée à la Figure 17. Les champs de record imprimés en gras sont les champs supplémentaires par rapport à l'implémentation de DZA&BLE alors que les champs en italique sont des champs inclus par DZA&BLE qui ne sont pas nécessaires dans l'implémentation que nous faisons.

```
PSon = ^TSon;
PItem = ^TItem;
PLayer = ^TLayer;
PInfo = ^TInfo;

{ element d'un arbre partagé }
TInfo = array[1..100] of integer;

{ arbre partagé }
SharingTree = record
              FirstLayer, LastLayer: PLayer;
              Root: PItem
              end;

{ niveau dans un arbre partagé }
TLayer = record
         FirstItem, LastItem: PItem;
         Previous, Next: PLayer
         end;

{ noeud dans un arbre partagé }
TItem = record
        Ident: integer;
        Info: integer;
        NbFathers: integer;
        NbSons: integer;
        FirstSon: PSon;
        NbElements: comp; { 64 bits integer }
        Marked: boolean;
        ToBeDev: boolean;
        First: boolean;
        Result: PItem;
        Previous, Next: PItem
        end;

{ fils d'un noeud dans un arbre partagé }
  TSon = record
         Son: PItem;
         Next: PSon
         end;
```

Figure 17 : Définition d'un arbre partagé en Pascal

Un arbre partagé est donc défini par sa racine (Root) et une liste de couches pointée par FirstLayer.

Chaque couche contient une liste d'éléments (les noeuds de l'arbre partagé) appelés Items et les pointeurs vers les couches précédentes et suivantes.

Chaque élément d'une couche (TItem) contient les champs d'information suivants :

- Un identificateur du noeud dans l'arbre : *Ident*.
- La valeur de l'état correspondant : *Info* (ici un entier, mais peut être de n'importe quel type).
- Le nombre de pères de l'élément dans l'AP : *NbFathers* (servira uniquement lors des optimisations).
- Le nombre de fils de l'élément : *NbSons*.

- Un pointeur vers la liste des fils : *FirstSon*.
- Le nombre d'éléments du sous-AP de racine ce noeud : *NbElements* (servira uniquement lors des optimisations).
- Un booléen *marked* (gardé pour la compatibilité avec DZA&BLE)
- Un booléen indiquant si le noeud doit être développé : *ToBeDev*.
- Un champ utilisé lors des comparaisons de sous-APs : *First*.
- *Result* : PItem (gardé pour la compatibilité avec DZA&BLE)
- Des pointeurs de chaînage des noeuds dans la couche : *Previous* et *Next*.

La Figure 18 donne un aperçu graphique de la structure d'un arbre partagé telle que définie ci-dessus.

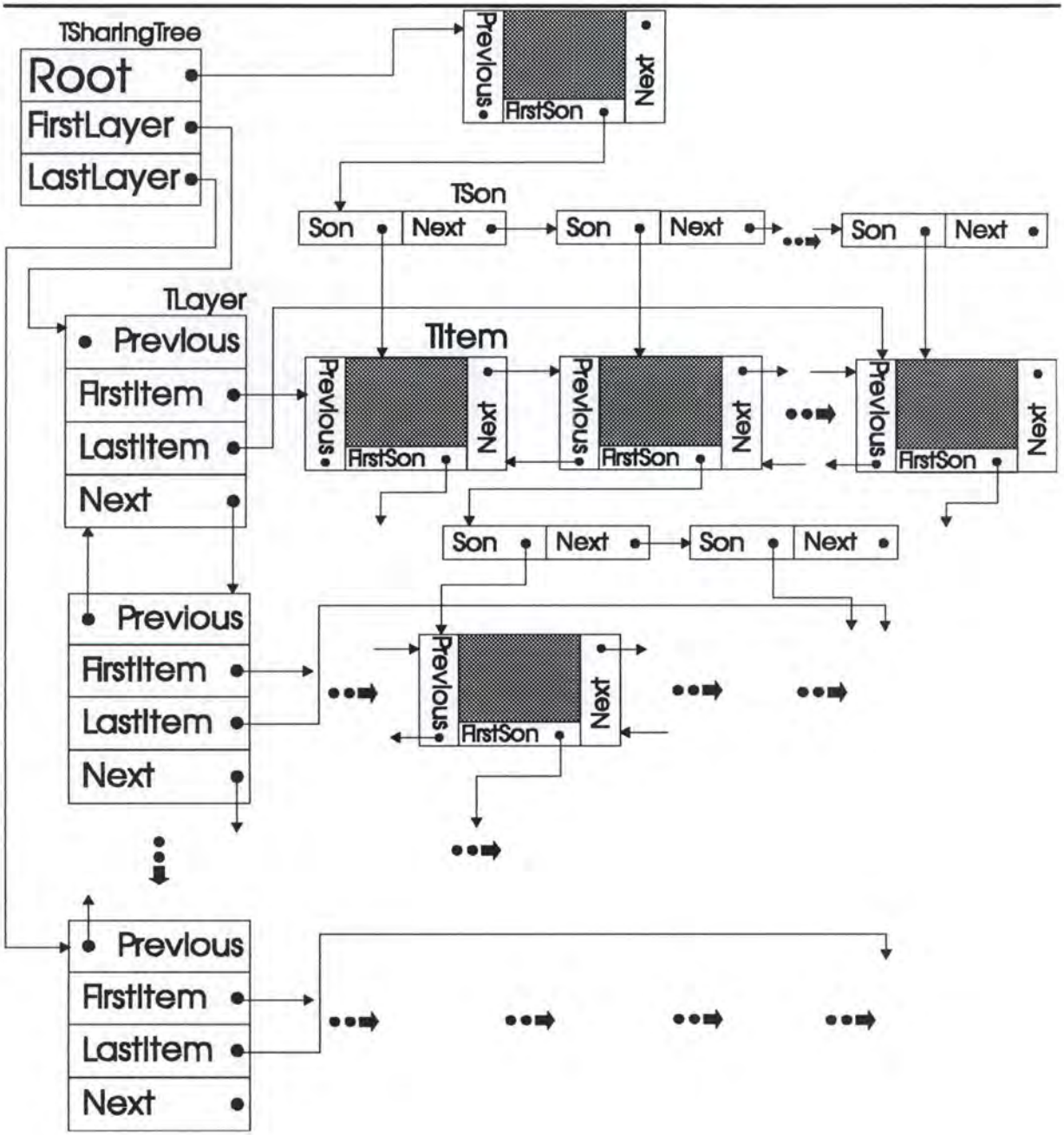

Figure 18 : Représentation graphique de la structure d'arbre partagé

5.4.3. Une structure de données pour la représentation des états à développer

Au départ, nous avions choisi de stocker les états à développer dans une structure de données du type suivant :

```
PItemList = ^TItemList;
TItemList = record
              item : PItem;
              next : PItemList;
            end;
```

Cette structure stockait donc non pas des états, mais des noeuds de l'arbre partagé. En effet, on a vu qu'il pouvait y avoir plusieurs noeuds représentant le même état sur une même couche (du moment qu'ils satisfaisaient les règles concernant les APs ou les ASPs). On ne développera donc plus des états comme auparavant mais bien des noeuds de l'ASP. Il était donc plus correct de parler de noeud à développer plutôt que d'état. D'où le stockage des noeuds à développer.

Cependant, cette structure s'est avérée comporter quelques failles. En effet, notre algorithme développe un arbre sous-partagé (pas un arbre partagé réel) et procède, comme nous le verrons dans la suite, à des réductions périodiques de l'ASP en AP. Or, la réduction entraîne inévitablement une non-stabilité des adresses des noeuds dans l'arbre (e.g. certains noeuds superflus sont supprimés lors de la réduction). Ce fait nous a obligés à changer de structure et à trouver un autre moyen d'identification des noeuds à développer.

Notre algorithme a la particularité de travailler toujours sur le développement des branches d'un père. Tout développement dans Transmit ou dans Propagate est rattaché à un et un seul noeud du niveau supérieur lorsqu'on sort de ces procédures. En effet, on remarque que dans Transmit on effectue un partitionnement en S et E_s alors que Propagate est soit appelée par ConstSync lorsque le niveau supérieur n'a pas encore été développé, soit par Transmit avec un ensemble de sous-états ayant même racine s.

De plus, on sait qu'il est possible d'identifier un noeud étant donné un de ses pères et l'état qu'il représente. En effet, on a vu lors de la définition d'un arbre partagé (ou d'un arbre sous-partagé) qu'un noeud ne peut avoir plusieurs fils de même valeur (c.-à-d. représentant le même état). Il est permis à tout moment d'identifier un noeud à développer par l'état qu'il représente.

Ces considérations nous ont donné la possibilité de représenter les noeuds à développer indépendamment de leur représentation physique. La structure de stockage des états à développer se présente donc de la manière exposée Figure 19.

```
PIntList = ^TIntList;
TIntList = record
             Info : integer;
             Next : PIntList;
           end;
```

Figure 19 : Représentation de la structure des noeuds à développer.

La Figure 20 montre l'identificateur d'un noeud à un moment donné. On voit qu'un noeud est identifié par son numéro et son père ("adresse de son père"). Il ne peut y avoir deux noeuds de même valeur ayant même père et, à tout moment du développement, on connaît le père des noeuds à développer.

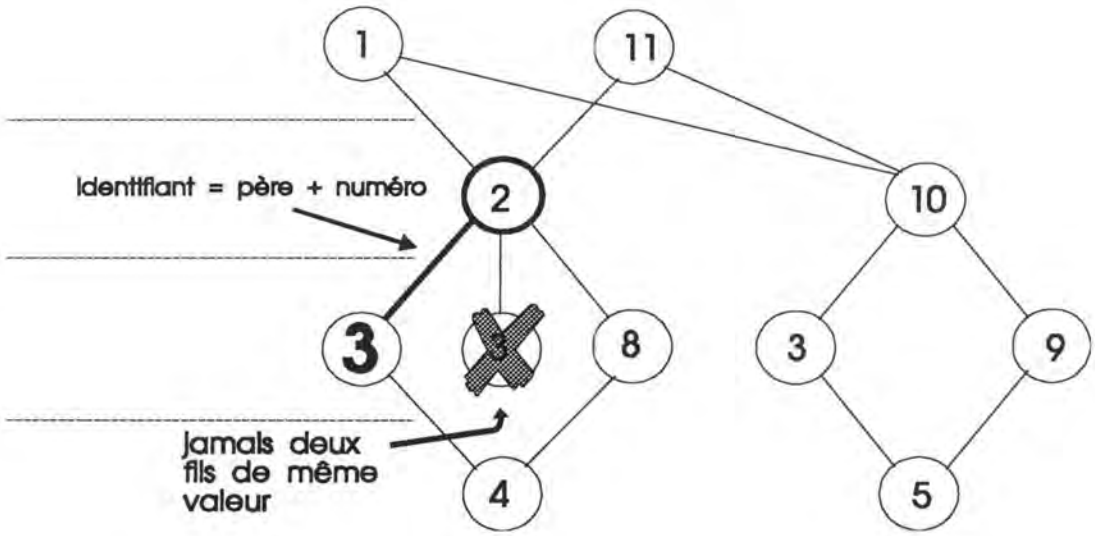

Figure 20 : Représentation graphique de la notion d'identifiant d'un noeud.

5.5. Les procédures principales

Dans ce paragraphe, nous montrons comment on a pu traduire l'algorithme théorique vu au paragraphe 4.2. Nous allons donc reprendre chaque procédure vue dans ce paragraphe et voir comment on peut y inclure l'idée d'arbre partagé. Nous verrons également comment on a pu implémenter les instructions non-standards des algorithmes théoriques.

5.5.1. La procédure Propagate

5.5.1.1. LES PARAMETRES

Le paragraphe 5.4.3 nous a expliqué la manière de stocker les noeuds à développer sous forme de liste d'états. Nous avons vu qu'il était nécessaire de connaître la racine du développement courant à tout moment de manière à pouvoir localiser un noeud à développer grâce à son père et l'état qu'il représente. Il est donc nécessaire de savoir quelle est cette racine (que nous appellerons père du développement de manière à ne pas confondre avec la racine de l'AP). D'où la nécessité d'un paramètre représentant le noeud "père du développement".

Comment représenter l'ensemble E de l'algorithme théorique ? Cet ensemble représente un ensemble de sous-états devant être reliés à un état unique du niveau supérieur (état connu au moment de l'appel). Il suffit donc de donner à la procédure le père du développement, puisque celui-ci est également la racine du sous-ASP représentant les sous-états du niveau à

développer. Lorsque *Propagate* est appelée via *ConstSync*, ce père est l'état initial du niveau précédent (ou la racine de l'ASP si on développe le niveau 1). Quand *Propagate* est appelée par *Transmit*, ce père est le père de l'ensemble des états se trouvant dans *slist* (élément créé lors des appels récursifs à la procédure *Transmit*).

En ce qui concerne la fonction *adevelop*, elle a été remplacée par le champ booléen *ToBeDev* dans les records de type *TItem*, qui détermine si un noeud est à développer.

Le niveau *niv* reste une variable entière. Cependant, il a été retiré des paramètres et transformé en variable globale de manière à épargner l'espace occupé par les paramètres des procédures récursives croisées *Propagate* et *Transmit*.

Les paramètres d'entrée de la procédure *Propagate* sont donc les suivants :

- father : PItem
  father est le père du développement à effectuer.
- slistptr : PItemList
  slistptr est un pointeur vers une liste qui contient l'ensemble des états à développer par la procédure.
- niv : integer
  niv est le niveau du développement (c'est l'indice de l'automate auquel appartiennent les états de slist).

Il n'y a pas de paramètre de sortie. En effet, les éléments ajoutés sont des fils de father. Comme *father* est un pointeur, il n'est pas modifié. C'est la structure sous-jacente qui est modifiée.

5.5.1.2. LE CORPS DE LA PROCEDURE

La documentation complète de la procédure Propagate a été incluse Annexe 1A.

La procédure est assez similaire à l'algorithme théorique. Les principales différences concernent la gestion de l'ensemble des tuples représentant les états atteints du produit synchronisé, les comparaisons d'ensembles transformées en comparaisons de sous-ASP et la représentation de ces différents ensembles par une racine commune à tous les tuples qu'ils représentent.

La gestion de l'ensembles des tuples consiste à vérifier à chaque ajout d'un noeud comme fils d'un autre dans une couche que les règles de cohérence des ASPs sont conservées. Cela revient à contrôler qu'aucun fils ne représente le même état que le fils que l'on veut ajouter à un père. C'est différents tests sont effectués grâce aux procédures de manipulation des arbres partagés qui sont décrites dans la section 5.6. En particulier la procédure AddDescenders permet d'ajouter à un sous-ASP de racine *s* l'ensemble des éléments d'un sous-ASP de racine s' (Très utile pour effectuer l'opération E := E U {s} x E').

Les comparaisons d'ensembles, c'est-à-dire les tests d'inclusion, sont effectués grâce à la fonction *Included*. Cette fonction utilise un mécanisme de comparaison de sous-ASP grâce au champ *First*, que nous exposons dans la section 5.6.

La représentation des différents ensembles sous la forme de sous-ASPs : les éléments représentés sont ceux décrits par la partie de l'ASP global incarnée par ce sous-ASP (c.-à-d. les tuples se trouvant sur tous les chemins partant de la racine).

```
Procedure Propagate(var slistptr:PIntList;father:PItem);

 var
   s,res : PItem;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;

 begin
   while slistptr <> nil do
     begin
      { Prendre l'état de début de liste et tronquer la liste }
      num := Head(slistptr);
      oldlist := slistptr;
      slistptr := Tail(slistptr);
      dispose(oldlist);
      { Recherche du noeud correspondant à l'état num }
      s := HasSon(father,num);
      { Remise à false du champ ToBeDev }
      s^.ToBeDev := false;
      with automate[niv]^.Etat[num]^ do begin
        { Traitement des transitions asynchrones }
        for j:=1 to nbSelfTransPoss do
          begin
            { Recherche de la destination de la jème transition }
            next := SelfTransPoss[j]^.destination;
            { Correspondant de l'opération E := E U {s'} X E'
              dans l'algorithme théorique }
            nextpitem := HasSon(father,next);
            { Traitement du cas où il n'existe pas encore de fils
              représentant l'état next ~ s' dans l'algorithme théorique}
            if nextpitem = nil then
              begin
                layerptr := GetLayer(PSync2,niv);
                nextpitem := AddItem(layerptr,next);
                nextpitem^.First := false;
                AddSon(father,nextpitem);
              end;
            { Test d'inclusion des descendants de s
              dans l'ensemble des descendants de nextpitem}
            if not(Included(s, nextpitem)) then
              begin
                { E := E U {s'} x E' }
                AddDescenders(s,nextpitem);
                { Ajout du nouvel état ou de l'état modifié à la liste }
                if not nextpitem^.ToBeDev then
                  begin
                    nextpitem^.ToBeDev := TRUE;
                    AddListItem(slistptr,next);
                  end;
              end;
          end {for};

        { Traitement des transitions synchrones }
        for j:=1 to nbSyncTransPoss do
```

```
begin
  { Recherche du niveau de synchronisation
    de la jème transition }
  syncniv := SyncTransPoss[j]^.synchronisateur;
  { Recherche de la destination de la
    1ère transition élémentaire }
  next := SyncTransPoss[j]^.destination;
  if (syncniv > niv) then
    begin
      { Création de l'élément père pour l'appel de Transmit }
      layerptr := GetLayer(PSync2,niv);
      res := AddItem(layerptr,next);
      res^.First := false;
      oldsource := sourceniv;
      sourceniv := niv;
      { Les adresses des noeuds ne sont pas stables dans
        la boucle, il faut donc recalculer l'adresse de s
        pour être sûr qu'elle correspond bien au noeud voulu}
      s := HasSon(father,num);
      { Appel de Transmit }
      transmit(syncniv,SyncTransPoss[j],s,res);
      sourceniv := oldsource;
      { Si au moins une transition synchrone
        a pu être effectuée, càd l'arbre créé n'est pas vide }
      if res<>nil then
        begin
          { Ajout du résultat à l'arbre partagé }
          nextpitem := HasSon(father,next);
          if nextpitem = nil then
            begin
              AddSon(father,res);
              res^.ToBeDev := true;
              AddListItem(slistptr,res^.Info);
            end
          else
            begin
              { Test d'inclusion du résultat dans le sous-arbre
                de racine le noeud fils existant nextpitem }
              if not (Included(res,nextpitem)) then
                begin
                  { Test de l'inclusion inverse pour optimisation
                    de la mise à jour }
                  if not (Included(nextpitem,res)) then
                    AddDescenders(nextpitem,res);
                  if nextpitem^.ToBeDev = true then
                    begin
                      nextpitem^.ToBeDev := false;
                      RemoveListItem(slistptr,nextpitem^.info);
                    end;
                  { Si nécessaire, retirer l'état représenté
                    par nextpitem de la liste car nextpitem
                    va être remplacé par res }
                  if nextpitem^.ToBeDev = true then
                    begin
                      nextpitem^.ToBeDev := false;
                      RemoveListItem(slistptr,nextpitem^.info);
                    end;
                  { Remplacement du fils nextpitem par res }
                  ReplaceSon(father,nextpitem,res);
                  { Mise à jour des champs First des éléments
                    concerné (cfr. traitement de l'inclusion }
                  AddFirst(nextpitem,res);
```

```
                                { Si le nbre de pères de nextpitem est nul
                                  on peut le supprimer de la couche }
                                if (nextpitem^.NbFathers = 0) then
                                  begin
                                    layerptr := GetLayer(PSync2,niv);
                                    DeleteItem(layerptr,nextpitem);
                                  end;
                                { Ajout du nouveau noeud à la liste }
                                res^.ToBeDev := true;
                                AddListItem(slistptr,res^.Info);
                              end;
                          end;
                      end {if res <> nil};
                  end {if syncniv > niv};
              end {for};
          end {with};
      end {while};
end;
```

5.5.2. La procédure Transmit

5.5.2.1. LES PARAMETRES

Comme pour la procédure *Propagate*, les ensembles ont été remplacés par des sous-ASPs. En conséquence, les paramètres E' et E'' sont remplacés respectivement par *s* (source de la première transition élémentaire de la transition synchrone à effectuer) et *res* (résultat : père du sous-ASP que produira *Transmit*).

De même, le paramètre *niv* a été transformé en variable globale pour les mêmes raisons que dans *Propagate*.

Le nom de la transition est remplacé par un pointeur vers sa description (de type PTransition) alors que le niveau synchrone à été calculé grâce à la structure d'automate et est un entier.

Cependant, la structure de stockage des transitions synchrones est telles qu'une transition synchrone est identifiée non seulement par le nom de cette transition mais également par le niveau avec lequel elle est synchrone. On pourrait donc avoir deux transitions synchrones ayant même nom mais synchrones avec des niveaux différents. Il est donc nécessaire pour les tests au niveau syncniv de savoir quel est le niveau source de la transition. Ce niveau peut être conservé dans une variable globale sourceniv, ce qui évite la saturation prématurée de la pile lors des appels récursifs.

Les paramètres d'entrée de la procédure Transmit sont donc les suivants :

- syncniv : integer.
  Niveau synchrone pour la transition considérée.
- trans : PTransition.
  Pointeur vers la description de la transition dans l'automate source.
- source : PItem.
  Racine du sous-ASP de base pour la transition (élément source de la première transition élémentaire).

- res : PItem.
  Racine du sous-ASP résultat de la transition (élément destination de la première transition élémentaire).

Tout comme dans Propagate, il n'y a pas de paramètre de sortie puisqu'on travaille uniquement avec des pointeurs.

5.5.2.2. LE CORPS DE LA PROCEDURE

Une description complète de la procédure Transmit est donnée en Annexe 1B.

```
procedure transmit(syncniv:integer;
                   sync:PTransition;source:PItem;var res:PItem);

 var
   son : PSon;
   slistptr : PIntList;
   j,next : integer;
   resprim : PItem;

 begin
   { Initialisation de la liste slistptr }
   slistptr := nil;
   { Si on est arrivé au niveau synchrone ...}
   if (niv+1) = syncniv then
     begin
       { Analyse de tous les fils de source }
       son := source^.FirstSon;
       while (son <> nil) do
         begin
           with automate[syncniv]^.Etat[son^.Son^.Info]^ do
             begin
               for j:=1 to nbSyncTransPoss do
                 begin
                   { Si la transition cherchée est trouvée ...}
                   if (SyncTransPoss[j]^.nom = sync^.nom) and
                      (SyncTransPoss[j]^.initiateur =
                                      (not (sync^.initiateur))) and
                      (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                     begin
                       { Recherche de la destination de la transition }
                       next := SyncTransPoss[j]^.destination;
                       { Recherche du noeud éventuel correspondant }
                       layerptr := GetLayer(PSync2,syncniv);
                       nextpitem := HasSon (res,next);
                       { Si le noeud n'existe pas, on le crée }
                       if nextpitem = nil then
                         begin
                           nextpitem := AddItem(layerptr,next);
                           AddSon(res,nextpitem);
                         end ;
                       nextpitem^.First := true;
                       { Test d'inclusion des descendant de son
                         dans les descendants de nextpitem }
                       if not(Included(son^.Son, nextpitem)) then
                         begin
                           oldniv := niv;
                           niv := syncniv;
                           { Ajout des descendants de son à nextpitem }
```

```
                                AddDescenders(son^.Son,nextpitem);
                                niv := oldniv;
                                { Insertion de l'état représenté par nextpitem
                                  dans slistptr, s'il ne s'y trouve pas encore }
                                if not nextpitem^.ToBeDev then
                                  begin
                                    nextpitem^.ToBeDev := TRUE;
                                    AddListItem(slistptr,next);
                                  end;
                              end;
                          end;
                      end;
                  end;
                son := son^.Next;
              end;
        end
      { Si on n'a pas encore atteint le niveau synchrone ...}
      else
        begin
          { Boucle sur l'ensemble des fils de source }
          son := source^.FirstSon;
          while son <> nil do
            begin
              layerptr := GetLayer(PSync2,niv+1);
              resprim := AddItem(layerptr,son^.Son^.Info);
              resprim^.First := true;
              niv := niv+1;
              { Appel de transmit récursivement }
              transmit(syncniv,sync,son^.Son,resprim);
              niv := niv-1;
              { Si le résultat de transmit n'est pas vide, ajouter
                le noeud resprim à slistptr }
              if resprim <> nil then
                begin
                  AddSon(res,resprim);
                  AddListItem(slistptr,resprim^.Info);
                  resprim^.ToBeDev := TRUE;
                end;
              son := son^.Next;
            end;
        end;
      { Si res possède un fils, on appelle Propagate pour développer le
        niveau auquel se trouvent les fils de res }
      if (res^.FirstSon <> nil) then
        begin
          niv := niv+1;
          Propagate(slistptr,res);
          niv := niv-1;
        end
      { Si res ne possède pas de fils, aucune transition n'a pu
        être effectuée.  On signale cela en renvoyant res = nil }
      else
        begin
          layerptr := GetLayer(PSync2,niv);
          DeleteItem(layerptr,res);
          res := nil;
        end;
    end;
```

La procédure théorique est divisée en deux cas :

- k = sync
- k < sync

On retrouve ces deux cas dans l'instruction

if (niv+1) = syncniv ...

En effet niv est le niveau (la couche) auquel se trouve le noeud source. Comme on considère les fils de ce noeud (dans la procédure théorique, on part toujours avec un k se situant un niveau en dessous du niveau courant), la comparaison se fait avec niv+1.

Considérons tout d'abord le cas (niv+1) = syncniv :

La structure d'arbre partagé va nous permettre d'éviter les opérations ensemblistes coûteuses de l'algorithme théorique. Il n'est plus nécessaire de rechercher les ensembles S et E_s. En effet, S a pour équivalent l'ensemble des fils du noeud *source* alors que les E_s correspondant sont tout simplement les descendants de ces fils respectivement.

Par contre, il n'est pas possible d'effectuer en une seule opération la création de E". Cette création doit se faire de manière itérative. On va donc pour chaque fils de *source* vérifier si c'est un candidat possible à une transition élémentaire synchrone (c'est-à-dire s'il y a une transition synchrone de valeur ~trans synchrone avec le niveau sourceniv). Cette phase est réalisée par une boucle while sur l'ensemble des fils de *source* dans laquelle on considère chaque transition synchrone partant de ce fils grâce à une boucle for.

Lorsqu'on trouve un noeud candidat à la seconde transition élémentaire de la transition synchrone, on va rechercher la valeur de l'état correspondant à la destination de cette transition et ensuite effectuer les mêmes opérations que pour une transition asynchrone de manière à ajouter à res un fils de valeur la destination de la transition et ajouter à ce fils l'ensemble des descendants du noeud candidat. Si cette opération a provoqué des modifications (*Included* avait renvoyé false ou bien le noeud n'existait pas encore), alors ce noeud doit être développé dans Propagate (on l'ajoute donc à la liste *slist*).

Quand toutes les transitions élémentaires valides ont été effectuées, si *slist* n'est pas vide (ce qui correspond en fait à *res^.FirstSon* <> *nil*), alors on appelle *Propagate* de manière à développer le niveau *syncniv* jusqu'à ce qu'aucune opération supplémentaire ne puisse être faite. Si *res* ne possède aucun fils, cela signifie qu'aucune transition élémentaire n'a pu être effectuée et dès lors le résultat renvoyé doit être *res* = *nil*.

Enfin on peut remonter d'un niveau en sortant de la procédure.

Voyons maintenant le cas *(niv+1)* < *syncniv* :

Cette partie est assez similaire à celle de l'algorithme. Elle consiste donc à appeler la procédure *Transmit* pour tous les fils de *source* de manière à descendre d'un niveau. Le résultat de la procédure *Transmit* (resprim dans le code) est ajouté comme fils de *res* et est inséré dans l'ensemble des noeuds à développer.

Lorsque la procédure *Transmit* a été appelée pour tous les fils, on peut développer les nouveaux noeuds en appelant *Propagate*. On n'appelle *Propagate* uniquement dans le cas où *res* possède au moins un fils. Dans le cas contraire, cela signifie qu'aucune transition élémentaire n'a pu être effectuée au niveau *syncniv* et le résultat renvoyé par la procédure doit être *res* = *nil*.

La Figure 21 nous montre l'état possible de l'ASP PSync après un appel récursif à *Transmit* dans un niveau intermédiaire. Les chiffres indiquent l'ordre de création des différents éléments lors de l'exécution de *Transmit*. En particulier, la sixième étape correspond à l'appel de *Propagate* dans *Transmit*. L'étape 10 est la prochaine action de la procédure et consiste à ajouter *resprim* renvoyé par *Transmit* lors de l'appel récursif comme fils de *res* qui à ce niveau de l'exécution est le noeud b.

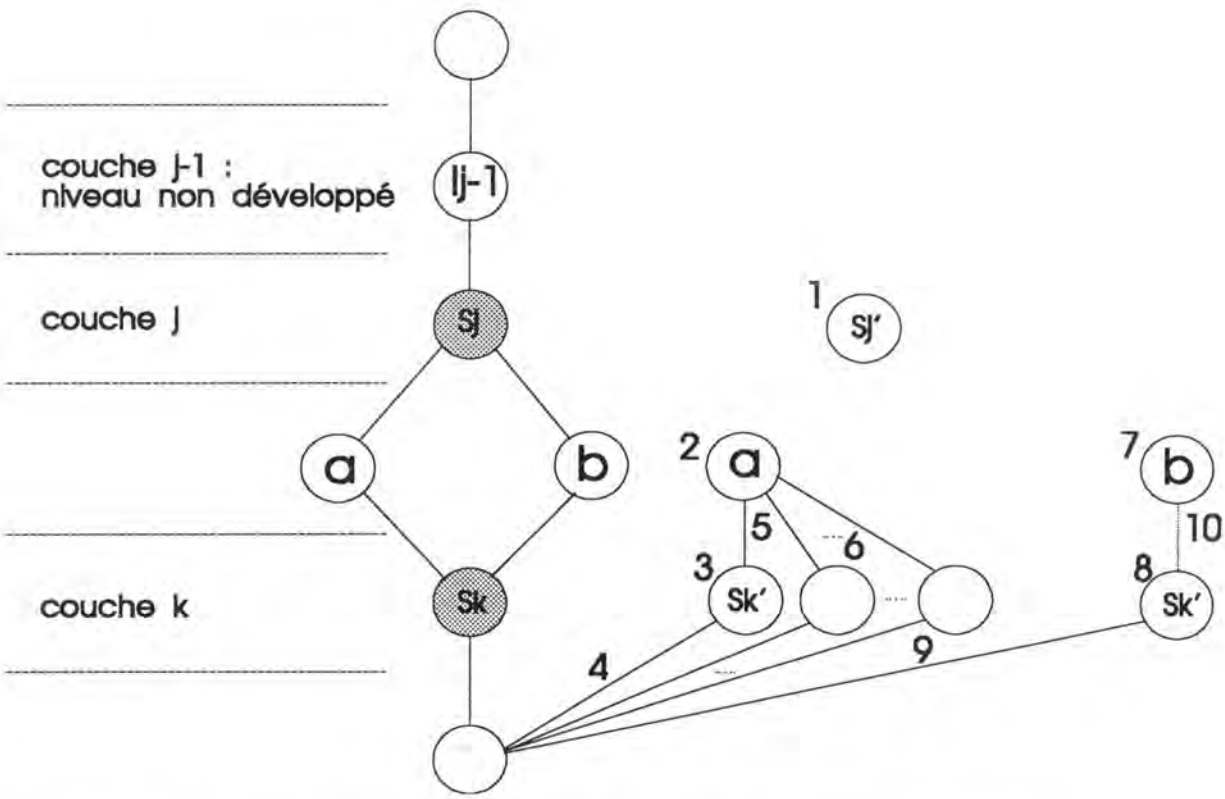

Figure 21 : Decription schématique de l'exécution de la procédure Transmit.

5.5.3. La procédure ConstSync

5.5.3.1. MODIFICATION DES STRUCTURES

L'utilisation des arbres partagés nous a obligé à modifier la structure de la procédure *ConstSync*. Dans l'algorithme théorique, on construit l'ensemble E en commençant par ses dernières composantes et en effectuant à chaque itération un produit cartésien avec le niveau supérieur. On ne peut appliquer cette technique lorsqu'on travaille avec des arbres partagés.

En effet, chaque composante correspond à une couche de l'AP. Or la gestion des couches se fait par ajout d'une dernière couche. Il n'existe pas de procédure d'insertion d'une couche entre deux autres. C'est pourquoi la procédure *ConstSync* que nous avons implémentée doit créer les premières couches avant de pouvoir développer les dernières.

Nous avons implémenté *ConstSync* sous la forme d'une procédure récursive. *ConstSync* crée une couche de l'AP, insère le noeud initial (de valeur 1) et l'ajoute comme fils du noeud initial du niveau précédent (pour le niveau 1, c'est la racine *Root* de l'arbre). Ensuite, si on ne se trouve pas au dernier niveau, on appelle *ConstSync* pour le niveau suivant. Enfin on appelle *Propagate* pour développer le niveau courant.

5.5.3.2. LES PARAMETRES

Pour fonctionner comme expliqué ci-dessus, la procédure ConstSync a besoin des données suivantes :

- result : PItem.
  Noeud du niveau précédent auquel doit être ajouté le résultat.
- i : integer
  Niveau courant.

Le nombre d'automates est donné par une variable globale nbautomates.

5.5.3.3. LE CORPS DE LA PROCEDURE

La procédure *ConstSync* est locale à la procédure *MySynchro* que nous décrivons dans la section suivante.

Elle est divisée en deux cas :

- i = nbautomates +1
- i < nbautomates +1

Lorsque $i = nbautomates + 1$, tous les niveaux de l'AP correspondant à des automates ont été créés. Il nous reste à créer l'élément fictif en ajoutant une couche supplémentaire et en y insérant un noeud de valeur (-MAXINT). Cet élément est ajouté comme fils de *result* (noeud créé au niveau supérieur) et on peut sortir de la procédure. A ce moment, l'arbre partagé contient un seul élément qui est l'état initial de l'automate produit.

Lorsque $i < nbautomates + 1$, il s'agit de créer une nouvelle couche dans laquelle on insère un noeud correspondant à l'état initial de l'automate Aut_i. Ce noeud doit être ajouté comme fils de *result* (racine de l'arbre si i=1 et noeud représentant l'état initial de Aut_{i-1} si $i \neq 1$). On appelle ensuite *ConstSync* avec $i+1$ et le nouveau noeud créé. Le résultat de cet appel récursif est le développement complet des niveaux inférieurs. Ensuite, on développe le niveau courant en insérant le noeud initial dans *slistptr* et en appelant la procédure *Propagate* avec *slistptr* et *result* qui est le père du développement (racine du sous-ASP résultat).

La description des procédure *ConstSync* et *Synchro* sont données à la **Erreur! Source du renvoi introuvable.**.

5.5.4. La procédure Synchro

Le but de cette procédure est l'initialisation de l'arbre partagé et l'appel de *ConstSync* pour le niveau $i = 1$ et avec la racine de l'AP comme paramètre *result*.

Cette procédure ne possède aucun paramètre puisque l'arbre partagé (*PSync2*) est déclaré comme variable globale, ainsi que toutes les informations concernant les automates.

La procédure procède donc à la création de l'AP PSync2 grâce à la procédure *STInit* et appelle ensuite *ConstSync*.

L'appel de *Synchro* permet donc la construction de l'ensemble des états accessibles du produit synchronisé.

```
procedure MySynchro;

var
  i     : integer;

  procedure ConstSync(i:integer; result:PItem);

  var
    slistptr : PIntList;
    item, partial : PItem;


  begin
    if i = nbautomates+1
    then begin
           { Création de la dernière couche de l'AP }
           layerptr := AddLayer(PSync2);
           { Insertion de l'élément fictif }
           item := AddItem(layerptr,-32767);
           item^.marked := false;
           item^.First := true;
           item^.ToBeDev := false;
           { Ajout de l'élément fictif comme fils de result }
           AddSon(result, item);
         end
    else begin
           { Création d'une couche supplémentaire }
           layerptr := AddLayer(PSync2);
           { Insertion du noeud correspondant à l'état initial }
           item := AddItem(layerptr,1);
           item^.marked := false;
           item^.First := true;
           item^.ToBeDev := false;
           { Ajout de ce noeud comme fils de result }
           AddSon(result, item);
           { Appel de ConstSync pour le niveau inférieur }
           ConstSync(i+1, item);
           { Insertion de item dans l'ensemble des noeuds à développer }
```

```
            item^.ToBeDev := true;
            new(slistptr);
            slistptr^.nitem := item^.Info;
            slistptr^.Next := nil;
            niv := i;
            { Réduction de l'ASP créé en AP }
            STReduce(psync2);
            { Appel de Propagate pour le développement du niveau courant }
            Propagate(slistptr,result);
          end;
  end;

begin
  i := 1;
  STInit(PSync2);
  ConstSync(i,PSync2.root);
end;
```

5.6. Spécification des procédures annexes

Dans cette section, nous donnons les spécifications de toutes les procédures et fonctions utilisées dans la section précédente.

Tout d'abord, nous exposons de manière succincte les procédures les plus importantes et en particulier celles que nous avons réalisées.

Ensuite, nous donnons une spécification des procédures de gestion des données et des procédures moins intéressantes d'un point de vue algorithmique.

5.6.1. Procédures et fonctions remarquables

5.6.1.1. LA FONCTION INCLUDED

Spécification :

En terme d'ensembles, la fonction a pour but de tester si la projection $\overline{proj_1}(E)$ (cfr. 3.1) d'un ensemble de tuples E est incluse dans la projection $\overline{proj_1}(E')$ d'un ensemble E'.

En terme d'arbres et de noeuds, la fonction *Included* revient à tester si l'ensemble des descendants d'un noeud n (racine du sous-ASP représentant E) est inclus dans l'ensemble des descendants d'un noeud n' (racine du sous-ASP représentant E'). En effet chaque noeud d'un arbre partagé identifie un ensemble de tuples représenté par le sous-ASP dont il est la racine.

Les paramètres de *Included* sont deux *PItem* (pointeurs vers des noeuds) *first* et *second*.

Conditions Initiales :

first et *second* sont des noeuds de même niveau dans un AP ou un ASP.

Conditions Finales :

- Les pointeurs *first* et *second* sont inchangés.
- *Included* = true si l'ensemble des descendants de *first* est inclus dans l'ensemble des descendants de *second*.
- *Included* = false dans le cas contraire.

Explication de la démarche

L'algorithme de base d'une telle fonction consiste à parcourir les arbres contenant l'ensemble des descendants du premier noeud et à vérifier que les noeuds trouvés se retrouvent bien dans le second. On réalise cela de manière récursive sur la structure de l'arbre principal comme indiqué par la Figure 22. A un niveau, on vérifie que *second* possède des fils de même valeurs que ceux de *first* et pour chaque fils de *first*, on appelle *Included* avec son homologue pour *second* de manière à vérifier l'inclusion aux niveaux inférieurs.

```
function Included(first,second: PItem) : boolean;

var
  son  : PSon;
  item : PItem;
  result : boolean;

begin
  { Boucle sur chaque fils de first }
  son := first^.FirstSon;
  result := true;
  while ((son <>nil) and (result)) do
    begin
      { Tester si second possède un fils de même valeur que son }
      item := HasSon(second,son^.Son^.Info);
      if item <> nil then
        { Test d'inclusion sur les descendants de son et item }
        result := Included(son^.Son,item)
      else
        { Si second ne possède pas de fils de même valeur que son,
          on n'a pas l'inclusion }
        result := false;
      son := son^.Next;
    end;
  Included := result;
end;
```

Figure 22 : Algorithme de base de la fonction Included.

Cette algorithme de base nécessite de parcourir tous les noeuds des sous-ASPs *first* et *second* parfois plusieurs fois. C'est donc une technique assez lourde, surtout lorsqu'on l'applique à des noeuds de niveaux supérieurs (vu que les tests se font sur tous les niveaux inférieurs).

C'est pourquoi nous avons recherché une méthode plus efficace pour ce test d'inclusion. Une première optimisation peut être de tester si les noeuds équivalents trouvés pour fils de *first* et *second* ne sont pas en fait les mêmes noeuds. Dans ce cas, il n'est pas nécessaire d'appeler

Included puisque les sous-ASPs sont les mêmes. Cette première optimisation n'est efficace que dans le cas où l'arbre est restreint (la probabilité que deux noeuds de même valeur soient égaux est grande dans ce cas). Cette optimisation est la seule valable dans le cas général.

Cependant, nous pouvons nous baser sur la manière dont on développe les sous-ASPs dans *Propagate* et dans *Transmit* pour en déduire une méthode de comparaison beaucoup plus efficace. Pour ce faire, on utilise le champ *First* (qui n'a rien à voir avec le paramètre *first* de la fonction) associé à chaque noeud. Ce champ booléen indiquera s'il faut ou non appliquer la fonction d'inclusion à ce noeud.

Le champ First sera mis à true lors de la création d'un noeud dans les cas suivants :

- Le noeud est le noeud initial du niveau donné (noeud créé dans *ConstSync*).
- Le noeud est créé dans *Transmit* pour l'exécution d'une transition synchrone.
- Lorsqu'on utilise la procédure *ReplaceSon* (cfr. infra), on veillera à mettre le champ *First* du nouveau fils à true si celui de l'ancien l'était. Sinon, on ne change rien.

Dans tous les autres cas, le champ First sera mis à false.

La propriété utilisée dans la comparaison de deux sous-ASP est la suivante :

Si l'ensemble des descendants marqués *First* d'un noeud est inclus dans l'ensemble des descendants d'un autre noeud, alors l'ensemble des descendants du premier noeud (y compris ceux qui ne sont pas First) sont inclus dans l'ensembles des descendants du second.

Les hypothèses de cette propriété sont que toutes les propriétés de l'arbre partagé en construction soit vérifiées. En particulier, on suppose que la comparaison se fait au niveau courant (niv). Ceci implique que tous les niveaux inférieurs ont été complètement développés.

Démonstration de la propriété :

Supposons par l'absurde que tous les descendants de *first* marqués First (dont le champ First est à true) soient inclus dans l'ensemble des descendants de *second* et qu'il existe un élément *e* dans l'ensemble des descendants de *first* dont au moins un noeud n'est pas marqué First et qui n'est pas inclus dans l'ensemble des descendants de *second*.

Nous allons montrer que dans ce cas, les sous-ASPs de racine second n'est pas complètement développé. En effet, considérons un noeud de *e* non marqué First. Cela signifie automatiquement que ce noeud a été créé dans *Propagate* (nous avons vu que les noeuds créés dans *Transmit* et dans *ConstSync* étaient marqué First). En fait, chaque développement dans *Propagate* est issu de un ou plusieurs noeuds marqués First (puisque *slist* contient à l'appel soit un élément créé dans *ConstSync* (le noeud initial du niveau) soit un ou plusieurs éléments créés dans *Transmit*).

Le noeud *e* non marqué First est donc la destination d'une suite de transitions de source un noeud marqué First (que nous noterons *f*). S.p.d.g. on peut supposer que cette suite est de longueur 1 (notons cette transition *t*).

Considérons un noeud *f'* appartenant à l'ensemble des fils de *second* homologues de *f* (le sous-ASP de racine *f* contient l'ensemble des éléments marqués First du sous-ASP de racine *f'*).

- Supposons tout d'abord qu'on ait *Included (f, f')* (inclusion de tous les éléments).

Dans ce cas, il n'est pas possible que le sous-ASP de racine *e* ayant mêmes descendants que *f* n'ait pas d'homologue dans les descendants de *second*. En effet, comme le sous-ASP *f* est inclus dans le sous-ASP *f'*, toutes les transitions possibles à partir des éléments du sous-ASP *f* le sont aussi à partir du sous-ASP *f'*. En particulier, la transition *t* à partir de *f* fournit un noeud *e'* racine d'un sous-ASP contenant l'ensemble des éléments du sous-ASP de racine e.

- Examinons maintenant le cas où on n'a pas *Included(f,f')* (certains éléments non marqués First ne sont pas inclus dans f').

On sait par hypothèse que tous les descendants de *f* marqués First sont inclus dans l'ensemble des descendants de *f'*. Donc *f* et *f'* vérifie bien les hypothèses de la propriété. On peut donc tenter une démonstration par l'absurde de la même manière sur *f* et *f'*. Cependant, comme à chaque fois que l'on se retrouve dans le cas *not Included (f, f')* on descend d'au moins un niveau et comme le nombre de niveaux est fini, on finira toujours (au pire on se retrouvera au dernier niveau) par se retrouver dans le cas *Included (f, f')* où l'on peut montrer l'absurdité (puisqu'au dernier niveau, tous les noeuds ont même descendant ⊥.

c.q.f.d.

Cette propriété nous permet lors des tests d'inclusion de ne considérer que les noeuds marqués First. On réalise cela en ajoutant un test sur la nature des noeuds examinés. La Figure 23 représente la fonction Included tenant compte des deux optimisations exposées.

```
function Included(first,second: PItem) : boolean;

var
  son  : PSon;
  item : PItem;
  result : boolean;

begin
  son := first^.FirstSon;
  result := true;
  while ((son <>nil) and (result)) do
    begin
      { On ne teste que les éléments marqués First }
      if son^.Son^.First then
        begin
          item := HasSon(second,son^.Son^.Info);
          if item <> nil then
            begin
              { On appelle Include récursivement uniquement
                si les noeuds considéré sont différents }
              if item <> son^.Son then
                result := Included(son^.Son,item);
            end
          else
            result := false;
        end;
      son := son^.Next;
    end;
  Included := result;
```

```
end;
```

Figure 23 : Algorithme de la fonction Included

5.6.1.2. LA PROCEDURE ADDGRAPH

Spécification :

La procédure *AddGraph* a pour but d'ajouter un sous-ASP à l'ensemble des descendants d'un noeud.

Paramètres d'entrée :

father, root : PItem.
rootlayer : PLayer.

Conditions initiales :

Le noeud pointé par *root* fait partie de la couche d'un ASP pointée par *rootlayer*.
Le noeud pointé par *father* fait partie du même ASP que celui pointé par *root* et est inclus dans la couche supérieure à celle de *root*.

Conditions finales :

Les pointeurs *father*, *root* et *rootlayer* sont inchangés.
Le sous-ASP de racine *root* a été ajouté à l'ensemble des descendants de *father*.

Explication de la démarche

L'algorithme de AddGraph consiste à partir de la couche pointée par *root* et à compléter l'ensemble des noeuds fils de *father* par *root*.

Si un noeud de même valeur existe déjà, alors plusieurs cas se présentent :

- Ce noeud est *root* lui-même (pointeurs de même valeur).
  Dans ce cas, il ne faut rien faire puisque le sous-ASP de racine *root* est inclus dans l'ensemble des descendants de *father*.
- Ce noeud n'est pas *root* et le nombre de pères de ce noeud est 1.
  S'il n'y a qu'un seul père, ce père est *father*. On peut donc se servir du noeud trouvé et ajouter à l'ensemble des descendants de celui-ci les sous-ASPs de racine les fils de *root* (à l'aide d'appels récursifs à *AddGraph*).
- Ce noeud n'est pas *root* et le nombre de pères de ce noeud est supérieur à 1.
  Dans ce cas, on ne peut procéder comme dans le cas précédent. En effet, ajouter des éléments à l'ensemble des descendants du noeud trouvé modifierait non seulement l'ensemble des descendants de *father*, mais également celui de tous les pères du noeud. C'est pourquoi on va créer un nouveau noeud dans la couche *rootlayer* équivalent au noeud trouvé (même valeur, mêmes fils) que l'on va substituer au noeud trouvé dans l'ensemble des fils de *father* (en utilisant la procédure *ReplaceSon*). Ensuite on se retrouve dans le cas

précédant (le nombre de pères du nouvel élément est 1) et on peut donc procéder de la même manière.

Si aucun noeud de même valeur n'existe, alors on ajoute *root* comme fils de *father* et on sort de la procédure.

La Figure 24 présente l'algorithme PASCAL de *AddGraph*.

```
procedure AddGraph(father,root: PItem; rootlayer: PLayer);

var
  item,newitem: PItem;
  s: PSon;

begin
  { Recherche d'un noeud de même valeur que root dans les fils de father }
  item:= HasSon(father,root^.Info);
  { Si aucun fils n'est trouvé, on ajoute root aux fils de father }
  if item = nil
  then AddSon(father,root)
  { Si un fils est trouvé et que ce fils est root, ne rien faire,
    Sinon ... }
  else if item = root
       then
       else begin
              { Si le noeud trouvé a plusieurs pères et que l'ensemble
                des fils de ce noeud ne contient pas tous les fils de
                root (auquel cas il ne faut rien faire) }
              if (item^.NbFathers > 1)
              and not ContainsSons(item,root)
              then begin
                     { Création d'un nouvel élément }
                     newitem:= AddItem(rootlayer,item^.Info);
                     newitem^.First := item^.First;
                     { Substitution de item et newitem }
                     ReplaceSon(father,item,newitem);
                     newitem^.First := item^.First;
                     s:= item^.FirstSon;
                     { Ajout des fils de item à newitem }
                     while s <> nil do
                     begin
                       AddSon(newitem,s^.Son);
                       s:= s^.Next
                     end;
                     item:= newitem
                   end;
              { Appel de AddGraph pour tous les fils de root }
              s:= root^.FirstSon;
              while s <> nil do
              begin
                AddGraph(item,s^.Son,rootlayer^.Next);
                s:= s^.Next
              end
            end
end;
```

Figure 24 : Algorithme de la procédure AddGraph

5.6.1.3. LA PROCEDURE ADDDESCENDERS

Spécification

Paramètres d'entrée :

father, item : PItem

Condition initiale :

father et *item* sont des pointeurs vers des noeuds d'une même couche.

Condition finale :

L'ensemble des descendants de *item* appartient à l'ensemble des descendants de *father*.

Explication de la démarche

On utilise la procédure *AddGraph* vue au paragraphe précédent. En effet, il suffit d'ajouter à l'ensemble des descendants de *father* les sous-ASPs de racine les fils de *item*.

La Figure 25 nous donne un aperçu de la manière dont on procède

```
procedure AddDescenders(father,item:PItem);

 var
   root      : PItem;
   son       : PSon;
   layer     : PLayer;

 begin
   son := father^.Firstson;
   root := son^.Son;
   layer := GetLayer(PSync2,niv+1);
   { Boucle sur l'ensemble des fils de father }
   while son<>nil do
     begin
       { Appel de AddGraph }
       root := son^.Son;
       AddGraph(item,root,layer);
       son := son^.Next;
     end;
 end;
```

Figure 25 : Algorithme de la procédure AddDescenders

5.6.1.4. LA PROCEDURE STREDUCE

Spécification

Cette procédure transforme un Arbre Semi-Partagé en Arbre Partagé.

Paramètre d'entrée :

ST : SharingTree

Condition initiale :

ST a été initialisé (cfr. procédure *STInit*).

Condition finale :

ST est sous la forme canonique.

Explication de la démarche

La réduction d'un ASP en AP consiste à imposer la troisième règle de consistance d'un arbre partagé à *ST*, à savoir :

$\forall\, 0 \le i \le k, \quad \forall\, n_1, n_2 \in N_i, \quad n_1 \neq n_2 \quad val(n_1) = val(n_2) \Rightarrow succ(n_1) \neq succ(n_2)$
Deux noeuds égaux dans la même couche ne peuvent avoir même ensemble de fils.

Pour ce faire, on va parcourir les couches de l'ASP de bas en haut. Lors de l'examen de chaque couche, on va détecter les noeuds redondants (noeuds de valeur et de successeurs identiques à un noeud déjà visité). Ces noeuds redondants vont être marqués, leur champ *result* va être assigné à l'adresse du noeud équivalent non marqué redondant et ils vont être placés dans une liste.

Lors de l'examen de la couche supérieure, on remplacera tous les fils redondants par les "originaux" (fils équivalents non marqués redondants). Lorsque tous les fils redondants des noeuds de la couche supérieure auront été remplacés (à la fin de l'examen de la couche supérieure), on pourra supprimer tous les noeuds redondants de la couche inférieure en parcourant séquentiellement la liste. L'utilisation d'une liste permet de ne parcourir chaque couche une seule fois. On peut ainsi détecter les redondances à un niveau i tout en supprimant celle déjà détectées au niveau i+1. Sans liste, on aurait été obligé de d'abord supprimer les redondances du niveau i+1 avant de supprimer les noeuds redondants au niveau i+1 par un second passage dans la couche i.

Cette méthode garantit qu'il n'y a aucune perte d'information, puisque tous les fils marqués redondants seront remplacés par un fils équivalent avant d'être supprimés.

La Figure 26 nous montre comment on peut mettre en oeuvre cette méthode dans un algorithme PASCAL. On peut remarquer le cas particulier de la première couche. Cette couche ne doit pas être examinée puisqu'il est impossible d'y trouver deux noeuds de même valeur (étant donné que ces noeuds ont tous même père "*root*"). On notera également la nécessité de gérer deux listes. La première contient l'ensemble des noeuds redondants de la couche inférieure (c'est la liste *L*) tandis que la seconde (*LL*) construite à chaque itération contient les noeuds redondants déjà rencontrés au cours de l'itération courante.

On peut montrer qu'après l'examen d'une couche i, l'ensemble des noeuds se trouvant dans les couches j (j > i) sont non redondants. Dès lors, après l'examen de la couche 1, l'ensemble des noeuds des couches j (j > 1) sont non redondants. Comme nous avons vu ci-dessus que les

noeuds de la couche 1 étaient non redondants par définition d'un ASP, on peut dire que quand la procédure finit, l'ensemble des noeuds de l'arbre ST sont non redondants et donc ST est un arbre partagé canonique.

```
procedure STReduce(var ST: SharingTree);

var
  layer: PLayer;
  item, pritem: PItem;
  s: PSon;
  L,LL,Lp: PList;

begin
  L:= nil;
  layer:= ST.LastLayer;
  { Itération sur les couches de bas en haut }
  while layer <> nil do
  begin
    LL:= nil;
    item:= layer^.FirstItem;
    { Boucle sur l'ensemble des noeuds de la couche }
    while item <> nil do
    begin
      { Modification de la couche inférieure en fonction des
        redondances trouvées à l'itération précédente }
      s:= item^.FirstSon;
      while s <> nil do
      begin
        if s^.Son^.NbSons = -1
        then ReplaceSon(item, s^.Son, s^.Son^.Result);
        s:= s^.Next
      end;
      { La couche 1 ne doit pas être réduite car les éléments
        de cette couche ont tous le même père "root" }
      if layer <> ST.FirstLayer
      then begin
             { Les noeuds de chaque couche sont classés par ordre
               croissant du champ Info.  Pour chaque noeud, on
               recherche un noeud de même valeur ayant même ensemble
               de fils et non marqué redondant (recherche
               nécessaire uniquement parmi les noeuds déjà visité }
           pritem:= item^.Previous;
           while pritem <> nil do
             if pritem^.Info <> item^.Info
             then pritem:= nil
             else if (pritem^.NbSons <> -1)
                  and SameSons(pritem, item)
                  then begin
                         { Si le noeud courant est redondant,
                           Alors on met NbSons à -1 (marquage),
                           on met dans le champ Result le pointeur
                           vers le noeud non redondant équivalent
                           et on place le noeud courant dans la liste
                           des noeuds redondants trouvés }
                       item^.NbSons:= -1;
                       item^.Result:= pritem;
                       new(Lp);
                       Lp^.Item:= item;
                       Lp^.Next:= LL;
                       LL:= Lp;
```

```
                        pritem:= nil
                      end
                else pritem:= pritem^.Previous
          end; {if}
      item:= item^.Next
    end; {while}
    { Effacement des noeuds redondants de la couche inférieure
      (se trouvant dans la liste L)}
    while L <> nil do
    begin
      Lp:= L^.Next;
      DeleteItem(layer^.Next, L^.Item);
      dispose(L);
      L:= Lp
    end;
    { Assigner la liste L à la liste des noeuds
      redondants de la couche courante }
    L:= LL;
    { Passer à la couche supérieure }
    layer:= layer^.Previous
  end
end;
```

Figure 26 : Algorithme PASCAL de la procédure de réduction d'un ASP en AP.

5.6.2. Autres procédures

5.6.2.1. GESTION DES ARBRES PARTAGES

`function ADDLAYER(var ST: SharingTree): PLayer;`

Cette fonction ajoute une couche à un AP ou ASP et renvoie le pointeur vers celle-ci. La couche ajoutée est la dernière de l'arbre (elle est également pointée par ST.LastLayer).

Paramètre d'entrée/sortie :

ST : SharingTree. AP ou ASP auquel la fonction s'applique.

Condition initiale :

ST a été initialisé (cfr. *STInit*)

Condition finale :

Une dernière couche a été ajoutée à l'AP (ou ASP) ST. Le pointeur vers cette couche est renvoyé par la fonction.

`function GETLAYER(var ST: SharingTree; n: integer): PLayer;`

Cette fonction renvoie un pointeur vers la $n^{ième}$ couche de l'arbre ST

Paramètre d'entrée :

n : integer. Entier représentant le numéro de la couche à rechercher.

Paramètre d'entrée/sortie :

ST : SharingTree. AP ou ASP sur lequel s'effectue la recherche.

Condition initiale :

ST a été initialisé (cfr *STInit*)

Condition finale :

Si l'arbre ST comporte n couches ou plus, la fonction renvoie le pointeur vers la $n^{ième}$ couche. Sinon elle renvoie le pointeur *nil.*

```
function AddItem(layer: PLayer; value: integer): PItem;
```

Cette fonction ajoute un noeud de valeur value à la couche layer et renvoie le pointeur vers ce nouvel élément.

Paramètre d'entrée :

layer : PLayer. Pointeur une couche d'un AP ou ASP.
value : integer. Valeur du noeud à insérer dans la couche.

Condition initiale :

layer <> nil.

Condition finale :

Un noeud de valeur value (dont le champ Info vaut value) a été ajouté à la couche pointée par layer et la fonction renvoie le pointeur vers ce noeud.

```
procedure DeleteItem(layer: PLayer; item: PItem);
```

Cette procédure efface un noeud d'une couche.

Paramètre d'entrée :

layer : PLayer. Pointeur vers la couche contenant le noeud à effacer.
item : PItem. Pointeur vers le noeud à effacer.

Conditions initiales :

layer <> nil.
item pointe vers un noeud appartenant à la couche pointée par layer.

Condition finale :

Le noeud pointé par item a été supprimé de la couche pointée par layer et l'espace mémoire a été libéré.

```
function HASSON(item: PItem; value: integer): PItem;
```

Cette fonction vérifie si le noeud item a un fils de valeur value. Si oui, la fonction renvoie le pointeur vers ce noeud, sinon elle renvoie nil.

Paramètre d'entrée :

item : PItem. Pointeur vers le noeud père.
value : integer. Valeur du noeud fils à rechercher.

Condition initiale :

item <> nil.

Condition finale :

La fonction renvoie *true* si le noeud pointé par *item* possède un fils de valeur *value*.
La fonction renvoie *false* dans le cas contraire.

```
function SAMESONS(i1, i2: PItem): boolean;
```

La fonction renvoie true si les noeuds pointés par i1 et i2 ont mêmes fils.

Paramètre d'entrée :

i1,i2 : PItem. Pointeurs vers les noeuds sur lesquels on effectue la comparaison.

Conditions initiales :

i1 <> nil, i2 <> nil.
i1 et i2 sont des pointeurs vers des noeuds de la même couche.

Condition finale :

La fonction renvoie true si i1 et i2 ont même ensemble de fils, false sinon.

```
procedure ADDSON(item, child: PItem);
```

Cette procédure ajoute le noeud pointé par child à l'ensemble des fils de item.

Paramètres d'entrée :

item : Noeud auquel le fils doit être ajouté.
child : Noeud fils à ajouter à item.

Conditions initiales :

item et child sont non nil.
item appartient à la couche immédiatement supérieure à celle de child.
item ne possède pas encore de fils de même valeur que child.

Condition finale :

item possède un fils supplémentaire pointé par child.

```
procedure REPLACESON(item, oldchild, newchild: PItem);
```

Cette procédure remplace le fils oldchild de item par le noeud newchild.

Paramètres d'entrée :

item : PItem. Pointeur vers le noeud "père".
oldchild, newchild : PItem. Resp. pointeurs vers l'ancien et le nouveau fils

Conditions initiales :

item <> nil, oldchild <> nil et newchild <> nil.
item a pour fils oldchild.
newchild est un noeud de la même couche que oldchild.

Condition finale :

Le noeud oldchild a été remplacé par newchild dans l'ensemble des fils de item.

```
function CONTAINSSONS(item1,item2: PItem): boolean;
```

Cette fonction renvoie true si l'ensemble des fils de item2 sont également fils de item1.

Paramètres d'entrée :

item1, item2 : PItem. Pointeurs vers des noeuds.

Conditions initiales :

item1 et item2 <> nil.

Condition finale :

La fonction renvoie true si l'ensemble des fils de item2 est contenu dans l'ensemble des fils de item1. Elle renvoie false sinon.

```
procedure ADDGRAPH(father,root: PItem; rootlayer: PLayer);
```

cfr. 5.6.1.2

```
procedure STINIT(var ST: SharingTree);
```

Cette procédure initialise la structure d'arbre partagé ST.

Paramètre d'entrée/sortie :

ST : SharingTree. Arbre partagé à initialiser.

Conditions initiales :

/

Condition finale :

$\mathcal{Elem}$(ST) = { }

```
procedure STEMPTY(var ST: SharingTree);
```

Cette procédure vide l'arbre ST.

Paramètres d'entrée :

ST : SharingTree. Arbre Partagé "à vider".

Conditions initiales :

ST a été initialisé.

Condition finale :

$\mathcal{Elem}$(ST) = { }

```
function STISEMPTY(var ST: SharingTree): boolean;
```

Cette fonction renvoie nil si l'AP ST ne contient aucun élément.

Paramètres d'entrée :

ST : SharingTree. Arbre partagé à tester.

Conditions initiales :

ST a été initialisé.

Condition finale :

STIsEmpty = ($\mathcal{Elem}$(ST) = { })

5.7. Les résultats obtenus

Nous avons testé l'algorithme de base sur de nombreux exemples classiques tels que les Schedulers de Milner, le problème du Producteur et du consommateur, le problème des philosophes, etc. Les conclusions sont que notre algorithme de base semble fonctionner de manière très satisfaisante. En effet, les temps d'exécution sont très proches et de ceux de l'algorithme DZA&BLE.

Le Tableau 3 reprend les temps des différents algorithmes connus et les compare à ceux obtenus avec notre algorithme de base. Le problème considéré est celui des Schedulers de Milner.

On remarque que les temps obtenus sont assez compétitifs. Il est cependant évident qu'il reste beaucoup à optimiser dans l'algorithme pour obtenir des temps comparables à ceux de DZA&BLE avec cache et pour pouvoir traiter des exemples plus importants que ceux-ci.

# automates	6	8	10	12	14	16	18	20
# états	577	3073	15361	73729	$34\ 10^4$	$15\ 10^5$	$7\ 10^6$	$31\ 10^6$
Fernandez	2.6	21	160	-	-	-	-	-
Groote	0.2	1.2	7.4	53	-	-	-	-
Enders	21	40	87	145	233	348	569	850
Rauzy	0.7	1.3	2.03	3.06	4.41	5.76	7.36	9.36
DZA&BLE[2]	0.28	0.99	6.59	85.3	-	-	-	-
JCC&co	0.22	0.98	5.66	26.75	-	-	-	-

Tableau 3 : Tableau comparatif des temps d'exécution pour le caclul des états accessibles d'un système d'automates formé par les schedulers de Milner

Nous avons vu au chapitre précédent les raisons pour lesquelles l'algorithme DZA&BLE pouvait être optimisé. Il s'agissait de ne pas effectuer plusieurs fois les mêmes opérations lors de la recherche du point fixe. Dans notre cas, il est probable également que certaines optimisations sont possibles.

[2] Les temps de DZA&BLE sont les temps de l'algorithme de base, sans cache.

Nous verrons dans la section suivante les points critiques de l'algorithme que nous avons exposé et les différentes optimisations qui ont été effectuées de manière à minimiser le temps d'exécution et nous verrons si ces optimisations parviennent à faire passer notre algorithme de temps exponentiels en fonction du nombre d'automates à des temps linéaires comme c'est le cas pour DZA&BLE.

6. Les optimisations apportées

L'algorithme de base ne constitue, comme son nom l'indique, qu'une ébauche de l'implémentation finale et nous allons examiner dans cette section l'ensemble des tentatives d'optimisation que nous avons faites. Nous verrons non seulement les optimisations efficaces mais également certaines qui ont échoué. En effet, c'est grâce aux conclusions des diverses tentatives que l'algorithme a pu être amélioré.

Les optimisations sont présentées dans l'ordre chronologique de l'évolution de l'algorithme. Ces optimisations ont toutes été réalisées à partir de l'algorithme de base. Celles-ci sont indépendantes et nous verrons dans quelle mesure il est possible de les combiner pour de meilleurs résultats.

A la fin de cette section, nous exposerons les possibilités d'optimisations que nous avons imaginées mais que nous n'avons pas implémentées, soit faute de temps, soit parce qu'elle exigeaient une dénaturation trop importante de l'algorithme de base.

6.1. Première tentative avortée : Même optimisation que DZA&BLE

La première idée que nous avons eue a été de reprendre le caching des opérations implémentées pour l'algorithme DZA&BLE. Cette optimisation permettait, comme nous l'avons vu au chapitre 2, d'éviter le recalcul de certaines opérations lorsque cela n'était pas nécessaire.

6.1.1. Rappel de la méthode

La méthode utilisée par DZA&BLE consiste à enregistrer lors de l'exécution d'une transition, les éléments suivants :

- *ident* : l'identificateur du noeud de départ de la transition.
- *nbFathers* : le nombre de pères du noeud au moment de l'enregistrement.
- *nbElem* : le nombre d'éléments au moment de l'enregistrement de l'ASP dont la racine est le noeud donné.
- *trans* : la transition effectuée après l'enregistrement.

Nous avons vu précédemment que ces éléments étaient nécessaires et suffisants pour déterminer si une transition devait être effectuée ou non (La transition devant être effectuée soit si le noeud source a plus de pères que précédemment, soit s'il a plus d'éléments dans son sous-arbre).

Ces informations sont stockées dans une table de hachage, ce qui permet de garantir un accès rapide aux données, moyennant une fonction de hachage adéquate.

6.1.2. Implémentation de la méthode.

6.1.2.1. DEFINITION D'UNE TABLE DE HACHAGE.

La table de hachage en elle même est uniquement un tableau dont les éléments sont des listes de record. La définition du tableau est la suivante :

```
type
  POpRecord = ^TOpRecord;
  TOpRecord = record
                IdIt: integer;
                NumE: comp;
                NumF: integer;
                IdTr: Ptransition;
                Next: POpRecord;
              end;

var
  OpTable: array[1..MaxOpTable] of POpRecord;
```

La gestion d'une table de hachage se fait au moyen de deux procédures. La première permet d'insérer des informations dans la table alors que la seconde permet de les retrouver. Pour ce faire, il est nécessaire de disposer d'une fonction de hachage qui calcule l'emplacement de l'information en fonction de son contenu. Dans notre cas, l'identifiant du noeud modulo la taille de la table + 1 est une valeur qui permet une répartition uniforme des informations sur la table. Le code des procédures de gestion se trouve en annexe 2.A.1. La spécification de ces dernières est donnée ci-dessous :

procedure RecordOper(Item: PItem; Trans: Ptransition)

Cette procédure insère dans la table de hachage les informations nécessaires à l'enregistrement de l'opération Trans sur Item.

Paramètres d'entrée :

Item : PItem. Noeud sur lequel l'opération doit être effectuée.
Trans : PTransition. Transition à effectuer.

Condition initiale :
/

Condition finale :

Les informations d'enregistrement ont été insérées dans la table.

function RecordedOper(Item: PItem; Trans: Ptransition): boolean

Renvoie True si l'enregistrement de l'operation Trans sur Item est présent dans la table de caching; False sinon

Paramètres d'entrée :

Item : PItem. Noeud sur lequel l'opération est effectuée.
Trans : PTransition. Transition à exécuter.

Condition initiale :
/

Condition finale :

S'il existe une entrée dans la table correspondant au résultat de l'opération RecordOper(Item,Trans), alors la fonction renvoie true; sinon elle renvoie false.

6.1.2.2. MODIFICATION DU CODE

Pour appliquer cette méthode à notre algorithme, certaines modifications sont nécessaires.

D'une part, nous avons inclus dans l'algorithme de base, les tests permettant de déterminer si une transition est utile ou non. Ces tests doivent être effectués avant chaque transition (synchrone et asynchrone) selon la même méthode que dans l'implémentation de DZA&BLE. Les modifications à ce niveau ne se font donc que dans la procédure Propagate puisque c'est dans cette dernière uniquement que les transitions sont initialisées (la procédure Transmit sert uniquement à exécuter une transition synchrone qui aura été jugée utile lors de l'initialisation dans Propagate).

D'autre part, le caching définit ci-dessous exige la connaissance du nombre d'éléments des sous-arbres de l'ASP en construction chaque fois que l'on veut utiliser la table de hachage.

La procédure STNbElements (dont l'algorithme se trouve en annexe 2.A.2) permet de calculer le champ NbElem de chaque noeud d'un arbre semi-partagé donné.

L'algorithme de base ne calcule pas le champ NbElem. Il est en effet assez coûteux de maintenir à tout moment cette information pour tous les noeuds de l'arbre. Cependant, il est primordial d'avoir cette information chaque fois, que soit on enregistre une opération dans la table de hachage, soit on recherche une information dans cette dernière.

Comme les accès à la table ne se font que dans la procédure Propagate, il n'est pas nécessaire d'introduire des modifications dans les autres procédures. Cependant, avant chaque transition dans Propagate, il faudra recalculer ce champ si l'arbre a été modifié depuis la dernière transition. On peut d'ores et déjà remarquer que ce calcul est assez pénalisant pour l'algorithme étant donné qu'il doit se faire assez souvent. Nous verrons dans une optimisation future comment il a été possible d'alléger ce temps de calcul en traitant l'arbre de manière locale et non globale.

La Figure 27 nous montre les parties du code de Propagate, modifiées pour y inclure l'optimisation définie ci-dessus.

```
Procedure Propagate(var slistptr:PIntList;father:PItem);

 var
   s,res : PItem;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;

 begin
   while slistptr <> nil do
     begin
       .
       .
       .
      with automate[niv]^.Etat[num]^ do begin
       { Traitement des transitions asynchrones }
       for j:=1 to nbSelfTransPoss do
         begin
           if not RecordedOper(s,SelfTransPoss[j]) then
             begin
               RecordOper(s,SelfTransPoss[j]);
               { Recherche de la destination de la jème transition }
               next := SelfTransPoss[j]^.destination;
               .
               .
               .
               { Test d'inclusion des descendants de s
                 dans l'ensemble des descendant de nextpitem}
               if not(Included(s, nextpitem)) then
                 begin
                   { E := E U {s'} x E' }
                   AddDescenders(s,nextpitem);
                   if not nextpitem^.ToBeDev then
                     begin
                       nextpitem^.ToBeDev := TRUE;
                       AddListItem(slistptr,next);
                     end;
                 end;
             end;
         end {for};

       { Traitement des transitions synchrones }
       for j:=1 to nbSyncTransPoss do
         begin
           if not RecordedOper(s,SyncTransPoss[j]) then
             begin
               { Recherche du niveau de synchronisation
                 de la jème transition }
               syncniv := SyncTransPoss[j]^.synchronisateur;
               .
               .
               .
             end;
         end {for};
      end {with};
    end {while};
 end;
```

Figure 27 : Modification de la procédure Propagate en fonction du caching des opérations

6.1.3. Les résultats obtenus

Le lecteur perspicace aura probablement déjà décelé la faille dans le raisonnement tenu dans ce paragraphe. En effet, si l'on observe les résultats obtenus, on remarque que le caching est inefficace et même inexistant. Les temps de calcul sont donc non pas diminués, mais bien augmentés vu les coûts engendrés par la gestion de la table et par le calcul des champs NbElem.

Comment expliquer cela ? Il faut se remémorer les conditions d'insertion dans la liste des noeuds à développer. On insère un noeud lorsqu'il est possible qu'une transition "utile" puisse être effectuée à partir de ce noeud. Nous avons vu qu'il y avait moyen de déterminer ces conditions d'insertion en utilisant la fonction Included dans Propagate (fonction qui détermine si on a inclusion entre deux sous-arbres, ce qui permet de voir si un sous-arbre est modifié ou non par une transition). De plus, dans Transmit et ConstSync, tous les noeuds insérés dans la liste sont des nouveaux noeuds et n'ont donc jamais été développés.

Les conséquences de cette technique d'insertion sont qu'aucun noeud n'est développé si le sous-ASP sous-jacent n'a été modifié au cours des opérations précédentes. Or le caching utilisé se base sur le fait qu'une transition ne doit pas être effectuée si le sous-ASP de racine le noeud origine n'a pas été modifié. Il est donc normal que ce caching soit inexistant.

6.1.4. Conclusion et conséquences

Il s'avère que ce caching ne peut s'appliquer à notre algorithme vu sa structure. On peut donc se demander pourquoi notre algorithme ne tourne pas plus vite vu qu'il ne possède pas la redondance de l'algorithme DZA&BLE.

Il est certain que certaines opérations redondantes sont également effectuées par notre algorithme. Cependant ces redondances sont d'un type différent. Elles ne sont pas directement liées à la manière dont l'algorithme détermine les noeuds à développer (comme c'est le cas dans DZA&BLE), mais plutôt à la manière dont l'arbre partagé est parcouru et construit.

Dans le paragraphe suivant, nous analyserons l'algorithme de base pour détecter les redondances dans les opérations et mettre au point un caching adapté à la structure de l'algorithme.

6.2. Deuxième tentative : Caching des transitions synchrones

6.2.1. Exposé de la méthode

Il apparaît assez vite qu'il est difficile d'envisager un caching sur les transitions asynchrones. En effet, il n'y a pas de redondance dans l'exécution de ces transitions, étant donné ce que nous avons vu dans le paragraphe précédent.

On ne peut pas tirer les mêmes conclusions pour les transitions synchrones. En effet, l'exécution d'une transition synchrone correspond comme nous l'avons vu précédemment, à la

création d'un nouveau sous-ASP. Cependant, si on analyse la manière dont ce nouveau sous-arbre est construit, on remarque qu'en fait, on crée non pas un sous-arbre, mais bien autant de sous-arbres qu'il existe de chemins entre le noeud source de la première transition élémentaire et le(s) noeud(s) source(s) de la seconde. Plus précisément, chaque fois que deux chemins valides se séparent, le sous-arbre en construction est divisé en deux parties correspondant aux deux chemins. La Figure 28 nous montre de manière graphique la technique adoptée. On remarque qu'il existe quatre chemins valides joignant les niveaux synchrones. Les parties encadrées sur la figure correspondent à deux sous-arbres identiques créés lors du parcours des chemins.

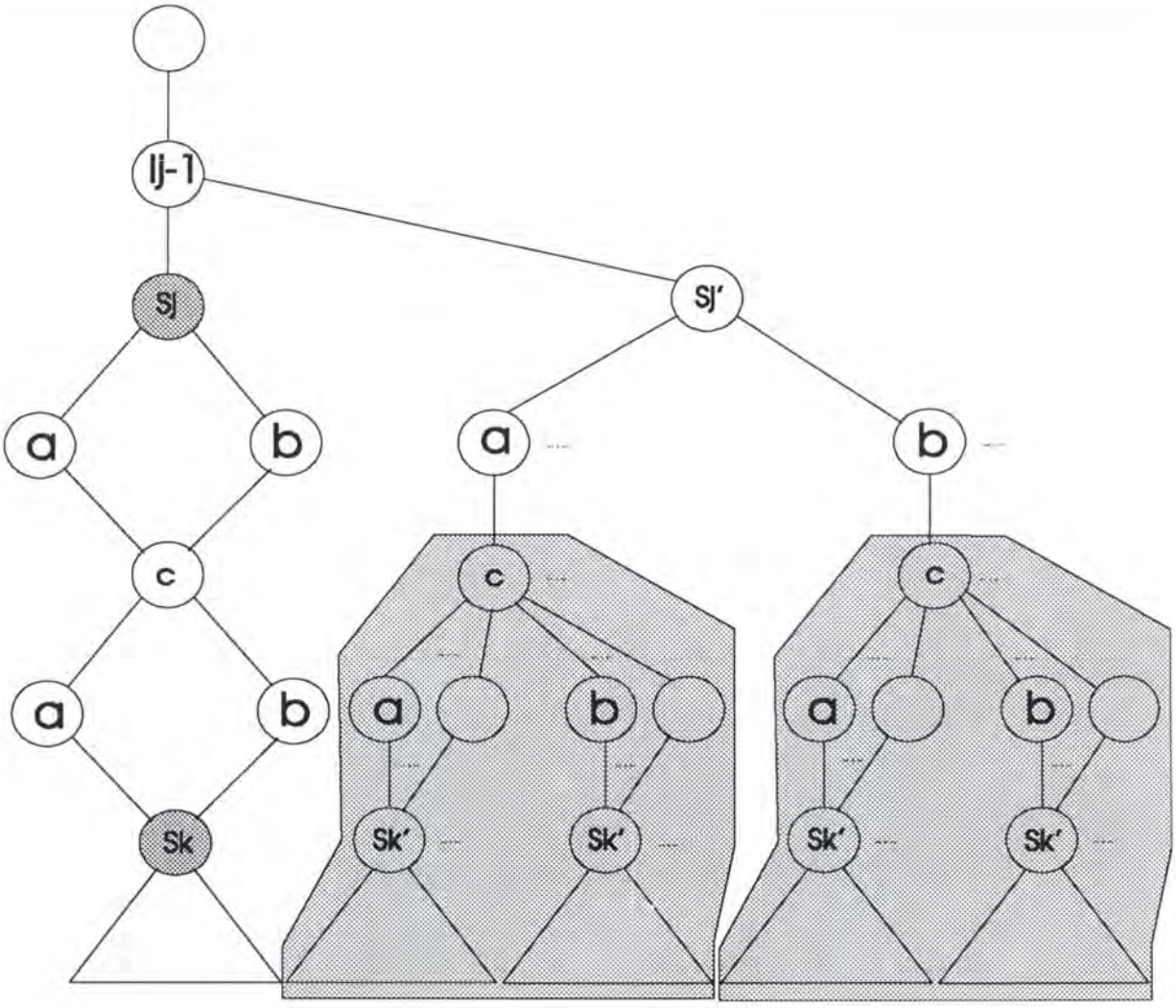

Figure 28 : Exécution de Transmit lorsqu'aucun caching n'est effectué.

Dans ce paragraphe, nous allons voir comment il est possible d'éviter cette redondance dans la structure lorsqu'on effectue les transitions synchrones. Il est important tout d'abord de réaliser pourquoi et quand on introduit cette redondance. On remarque que les parties redondantes correspondent à des parties de chemins valides communes. En effet, lorsque deux chemins se séparent, deux branches correspondantes sont créées. Cependant, même si ces chemins se rejoignent, les sous-arbres correspondants restent eux disjoints, d'où la redondance structurelle.

Le caching que nous allons implémenter aura donc pour but de détecter si deux chemins ne se rejoignent pas à un moment (c.-à-d. si un chemin ne passe pas par un noeud déjà emprunté pour la même transition synchrone). Auquel cas, on recherchera l'arbre correspondant de manière à éviter la reconstruction .

La méthode adoptée consiste à stocker lors de l'exécution d'une transition synchrone, les informations suivantes dans une table de hachage :

- IdentSource : Identificateur du noeud se situant sur un chemin.
- NumES : Nombre d'éléments du sous-ASP de racine Source.
- IdTr : Identificateur de la transition en cours.
- IdentDest : Identificateur du noeud créé lors de l'exécution de la transition.
- NumED : Nombre d'éléments du sous-ASP de racine Dest.

En effet, il est nécessaire, tout comme dans le caching de DZA&BLE, de savoir si un sous-arbre a été modifié depuis l'enregistrement du noeud racine. Comme on enregistre deux sous-arbres comme information pour le caching, on enregistre également le nombre d'éléments de ces sous-arbres dans les champs NumES et NumED.

Avant la création d'un noeud lors de l'exécution de Transmit, on vérifiera donc qu'un noeud équivalent n'a pas déjà été créé pour la même transition. Si c'est le cas, il suffira de raccorder le sous-arbre déjà construit au noeud équivalent. Sinon, on exécutera la procédure de manière classique et on enregistrera les informations nécessaires pour pouvoir détecter lors d'un autre passage que le noeud a déjà été atteint.

6.2.2. Implémentation de la méthode

Bien que le principe soit assez simple, l'implémentation d'une telle méthode comporte quelques difficultés. En effet, deux problèmes se posent :

D'une part, il faut maintenir à tout moment le nombre d'éléments des différents sous-arbres de l'ASP en construction. La méthode de recalcul de ces champs pour l'arbre entier est assez lourde, surtout lorsque le caching se fait très souvent. Nous verrons qu'il y a moyen de maintenir ce champ correct par un recalcul local au niveau du noeud ou au niveau de la couche.

D'autre part, il faut déterminer le moment adéquat pour l'enregistrement du caching des opérations. On distinguera le cas du niveau synchrone et celui des niveaux intermédiaires.

6.2.2.1. LE CHAMP NBELEM

Deux techniques sont envisageables. Soit on considère le champ NbElem uniquement quand on en a besoin. Dans ce cas, un recalcul de ce champ pour tout l'arbre sera nécessaire. Soit on prend en compte le champ NbElem à tout moment, et alors il est possible d'utiliser des techniques moins lourdes pour maintenir ce champ à sa bonne valeur à tout moment.

Etant donné que notre caching est utilisé assez souvent dans la procédure Transmit, nous avons tenté d'implémenter une méthode de recalcul dynamique de ce champ à chaque modification pour les noeuds concernés. Ceci a été possible parce que notre algorithme travaillait localement sur l'ASP en construction. Ainsi, il suffit de ne tenir compte que de la couche inférieure pour maintenir le champ NbElem correct. En effet, ce nombre d'éléments peut être calculé de la manière suivante :

Le nombre d'éléments d'un sous-ASP de racine r est égal à la somme du nombre d'éléments des sous-ASPs de racine les fils de r.

L'utilisation de cette formule implique que l'on connaisse le nombre d'éléments des sous-ASPs de racine les fils de r. Comme l'algorithme travaille en développant entièrement un niveau avant de remonter, il va être possible de garantir à un niveau i que le champ *NbElem* de tous les noeuds de niveau j (j>i) est correct. En effet, au dernier niveau, le nombre de ces éléments est 1 (le noeud fictif créé lors de l'initialisation dans *ConstSync*) et en remontant dans chaque niveau, on peut évaluer les noeuds en fonction de leurs fils.

On introduit une procédure remplaçant la procédure *STNbElem*. Cette procédure (appelée ItemNbElem) permet de calculer le champ *NbElem* d'un noeud en fonction de ses fils.

Voici comment *ItemNbElem* est utilisée :

ItemNbElem est appelée après chaque modification du sous-ASP d'un noeud. Cela survient lorsqu'on utilise une des procédures suivantes :

- *AddSon*
- *AddDescenders* (ItemNbElem doit être utilisée dans AddGraph)
- *AddItem*
- *DeleteItem*
- *Propagate*
- *Transmit*

On ne s'occupe jamais des niveaux supérieurs. Ce n'est pas nécessaire puisque le calcul à ces niveaux se fait en remontant dans les appels des différentes procédures. Par exemple, le paramètre *father* de la procédure *Propagate* ne doit pas être mis à jour. Lorsqu'on sort de la procédure, et qu'on revient dans *ConstSync* ou dans *Transmit*, alors le noeud correspondant est mis à jour. On s'occupe donc uniquement des noeuds du niveau courant.

Le code des procédures *ConstSync*, *Propagate* et *Transmit* modifié pour y inclure ce calcul dynamique du champ NbElem de chaque noeud est donné en annexe 2.A.3 (ce code comprend également l'optimisation que nous sommes en train de voir).

6.2.2.2. CACHING DES TRANSITIONS SYNCHRONES

6.2.2.2.1. La table de hachage

On définit la table de hachage comme suit :

```
type
  PMyOpRecord = ^TMyOpRecord;
  TMyOpRecord = record
                  IdSource : integer;
                  NumES    : comp;
                  IdTr     : PTransition;
                  Destptr  : PItem;
                  NumED    : comp;
                  Next     : PMyOpRecord;
                end;
var
    MyOpTable : array[1..MaxOpTable] of PMyOpRecord;
```

Les procédures et fonctions de manipulation sont plus importantes que pour le caching de DZA&BLE. En effet, nous devons non seulement pouvoir enregistrer et tester si quelque chose à été enregistré, mais encore retrouver le résultat de l'opération (noeud pointé par destptr) s'il existe. Ci-dessous nous donnons la spécification des procédures et fonctions utilisées. Les détails d'implémentation se trouvent en annexe 2.A.1.

```
function MyRecordedOper(Item: PItem; Trans: Ptransition; NumESOK : boolean)
                      : PMyOpRecord;
```

Cette fonction renvoie True si l'enregistrement de l'opération Trans sur Item est présent dans la table de caching; False sinon

Paramètres d'entrée :

Item : PItem. Noeud sur lequel l'opération est effectuée.
Trans : PTransition. Transition effectuée sur le noeud Item.
NumESOK : Boolean. Paramètre qui indique s'il faut tester le champ NumES.

Condition initiale :

La table de hachage *MyOpTable* a été initialisée.
Item <> nil

Conditions finales :

Si *NumESOK* = True, alors, si une opération Trans sur Item a été enregistrée dans la table, on teste si *NumES* vaut bien *Item^.NbElem*. Si oui, on renvoie le pointeur vers la cellule contenant les informations, sinon on renvoie *nil*.
Si *NumESOK* = false, alors on ne vérifie pas *NumES* et on renvoie le pointeur vers la cellule contenant les informations si une cellule a été trouvée; sinon on renvoie *nil*.

```
procedure MyRecordOper(Item: PItem; Trans: Ptransition; DestItem: PItem);
```

Cette fonction enregistre l'operation Trans effectuée sur Item dans la table de hachage.

Paramètres d'entrée :

Item : PItem. Noeud source de l'opération.
Trans : PTransition. Transition effectuée au moment de l'enregistrement.
DestItem : PItem. Noeud destination de l'opération.

Condition initiale :

La table MyOpTable a été initialisée.
Item <> nil

Conditions finales :

S'il existe déjà une cellule dans la table *MyOpTable* représentant une opération faite sur *Item* par *Trans* (MyRecordedOper(Item,Trans,false) → True), alors le contenu de celle-ci est remplacé par l'information sur l'opération enregistrée. Sinon, une cellule contenant cette information est insérée dans la table.

```
function GetOpResult(Item:PItem; Trans: Ptransition) : POpres;
```

Cette fonction renvoie le "résultat" de l'opération Trans sur Item.

Type utilisé :

```
type
  POpres = ^TOpres;
  TOpres = record
             opOK : Boolean;
             item : PItem;
           end;
```

Paramètres d'entrées :

Item : PItem. Noeud source de l'opération *Trans* sur *Item* recherchée.
Trans : PTransition. Transition effectuée sur le noeud *Item*.

Condition initiale :

La table *MyOpTable* a été initialisée.
Item <> nil.

Condition finale :

Si l'enregistrement d'une opération *Trans* sur *Item* a été trouvée dans la table et si le résultat de cette opération (un sous-ASP) n'a pas été modifié depuis l'enregistrement (test sur *NumED*), alors la fonction renvoie un pointeur vers un *TOpRes* où *opOK* = true et item est la destination de l'opération (noeud *Destptr* dans l'entrée de la table de hachage).

Si aucun enregistrement n'a été trouvé ou si les informations correspondantes sont obsolètes alors la fonction renvoie un pointeur vers un TOpRes où opOK = false et item = nil.

6.2.2.2.2. Modification du code de la procédure Transmit.

Condidérons tout d'abord l'enregistrement des opérations.

Il faut distinguer deux cas :

D'abord le cas où on se trouve au niveau synchrone. Il faut alors enregistrer l'opération qui correspond à la seconde transition élémentaire. Cette opération se fait sur les fils du paramètre source et consiste à créer un ou des nouveaux fils à *res* et ensuite leur ajouter les descendants des fils correspondants de source. C'est seulement après que l'on peut l'enregistrer cette opération. En effet, sinon l'opération enregistrée serait immédiatement obsolète vu l'ajout des descendants qui modifient le sous-arbre.

Le second cas est celui où l'on se trouve dans un niveau intermédiaire. On sait que la technique adoptée est d'appeler récursivement *Transmit* avec tous les fils du paramètre source. On crée ainsi des éléments *resprim* qui correspondent à ces fils. On enregistre l'opération après l'appel de *Transmit*. Cela garantit que les sous-arbres de racine *resprim* ont été complètement développés.

La Figure 29 nous montre graphiquement ce qui est stocké lors de l'enregistrement d'une opération dans un niveau intermédiaire. Les deux sous-arbres grisés sont donc stockés avec la transition correspondante sous la forme expliquée précédemment. Les parties dessinées en gris clair sont celles qui n'ont pas encore été développées. On remarque parmi celles-ci que le noeud b qui sera créé sera attaché à l'arbre enregistré au lieu de créer un arbre supplémentaire. On peut aussi voir que la technique de caching a été appliquée au dernier niveau. Cette figure est à comparer avec la Figure 28 qui représente la même transition lorsqu'aucun caching n'est utilisé.

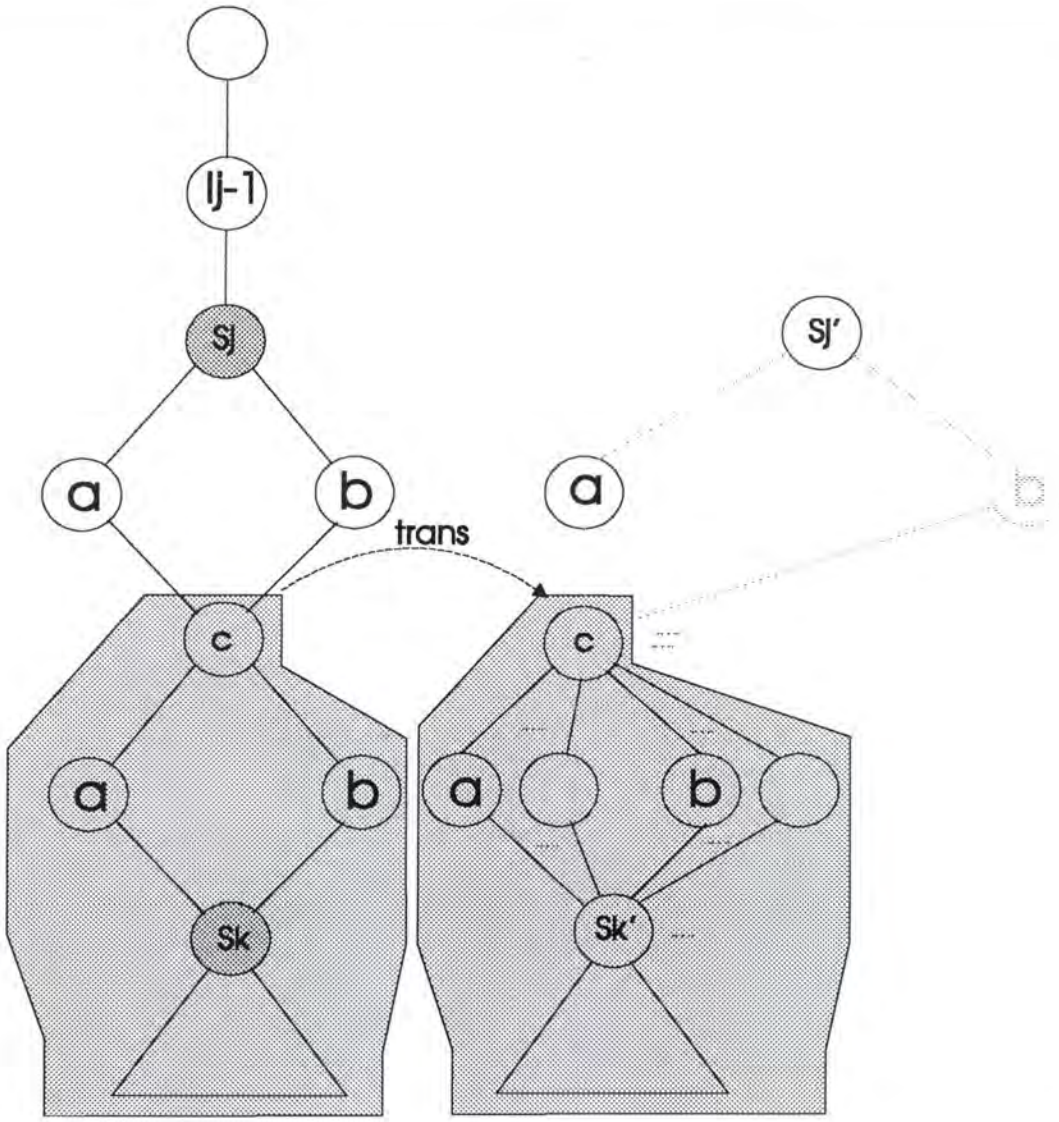

Figure 29 : Exemple d'application du caching des transitions synchrones.

En ce qui concerne la récupération des informations, c'est-à-dire le fait de retrouver le noeud correspondant à une exécution précédente de la même transition, on procède de la manière suivante :

Si l'on se trouve au niveau synchrone, alors, dès qu'un noeud fils de source à partir duquel on peut effectuer la transition élémentaire est trouvé, on va tester si on a déjà effectué cette transition auparavant. Pour ce faire, on utilise la procédure *GetOpResult* avec comme paramètre le noeud donné ainsi que le pointeur vers la transition synchrone du niveau supérieur (première transition élémentaire).

Si le résultat renvoyé (de type *POpRes*) est positif (*opOK* = true), alors le caching peut être appliqué. Il suffit d'ajouter à l'ensemble des fils de *res* le noeud pointé par le champ *item* du résultat. Sinon, on applique la procédure comme précédemment. Dans tous les cas, les noeuds créés ou retrouvés doivent être insérés dans la liste des noeuds à développer.

Lorsqu'on se trouve dans un niveau intermédiaire, on effectue les tests sur les fils de source. Si une opération de type *trans* a été enregistrée sur un fils de *res* et si les composants de cette opération n'ont pas été modifiés depuis, alors il suffit d'ajouter à l'ensemble des fils de *res* le noeud résultat de cette opération (pour déterminer cela, on utilise la fonction *GetOpResult*). Sinon, on appelle *Transmit* récursivement comme auparavant. Dans les deux cas, le nouveau fils est ajouté à la liste des noeuds à développer.

La Figure 30 nous montre la procédure Transmit telle qu'elle est lorsqu'on a introduit les optimisations sus-citées.

```
procedure transmit(syncniv:integer;
                   sync:PTransition;source:PItem;var res:PItem);

 var
   son : PSon;
   slistptr : PIntList;
   j,next : integer;
   resprim : PItem;

 begin
   slistptr := nil;
   if (niv+1) = syncniv then
     begin
       son := source^.FirstSon;
       while (son <> nil) do
         begin
           with automate[syncniv]^.Etat[son^.Son^.Info]^ do
             begin
               for j:=1 to nbSyncTransPoss do
                 begin
                   if (SyncTransPoss[j]^.nom = sync^.nom) and
                      (SyncTransPoss[j]^.initiateur =
                                        (not (sync^.initiateur))) and
                      (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                     begin
                       if opresult <> nil then
                         dispose (opresult);
                       if not RecTrans then
                         begin
                           { Si l'utilisateur a spécifié sans caching,
                             on met le résultat de la recherche à
                             aucune information trouvée }
                           new(opresult);
                           opresult^.item := nil;
                           opresult^.opOK := false;
                         end
                       else
                           { Recherche d'une opération précédente }
                           opresult := GetOpResult(son^.Son,sync);

                       nextpitem := opresult^.item;
                       next := SyncTransPoss[j]^.destination;
                       { Si la recherche a échoué ...}
                       if not opresult^.opOK then
                         begin
                           layerptr := GetLayer(PSync2,syncniv);
                           nextpitem := HasSon (res,next);
                           if nextpitem = nil then
```

```
                        begin
                          nextpitem := AddItem(layerptr,next);
                          AddSon(res,nextpitem);
                        end;
                      .
                      .
                      .
                      { Enregistrement de l'opération effectuée }
                      if RecTrans then
                        MyRecordOper(son^.Son,sync,nextpitem);
                    end
                  { CACHING !!! }
                  else
                    if nextpitem <> nil then
                      begin
                        { Ajout du noeud trouvé aux fils de res }
                        AddSon(res,nextpitem);
                        if not nextpitem^.ToBeDev then
                          begin
                            nextpitem^.ToBeDev := TRUE;
                            AddListItem(slistptr,next);
                          end;
                      end;
                end;
            end;
        end;
      son := son^.Next;
    end;
  end
else
  begin
    son := source^.FirstSon;
    while son <> nil do
      begin
        layerptr := GetLayer(PSync2,niv+1);
        resprim := AddItem(layerptr,son^.Son^.Info);
        resprim^.First := true;
        if opresult <> nil then
          dispose(opresult);
        if not RecTrans then
          begin
            { Si aucun caching demandé,mettre le résultat à
              aucune opération trouvée }
            new(opresult);
            opresult^.opOK := false;
            opresult^.item := nil;
          end
        else
            { Recherche du résultat d'une éventuelle
              opération enregistrée }
            opresult := GetOpResult(son^.Son,sync);
        nextpitem := opresult^.item;
        { Si la recherche a échoué ...}
        if not opresult^.opOK then
          begin
            niv := niv+1;
            transmit(syncniv,sync,son^.Son,resprim);
            niv := niv-1;
            { Enregistrement de l'opération }
            if RecTrans then
              begin
                if resprim <> nil then
                  ItemNbElem(resprim);
```

```
                    MyRecordOper(son^.Son,sync,resprim);
                  end;
                .
                .
                .
              end
            { Si la recherche donne un résultat positif ... CACHING !!!}
            else
              if nextpitem <> nil then
                begin
                  AddSon(res,nextpitem);
                  if nextpitem^.First = false then
                    begin
                      writeln('nextpitem^.first=false a la fin');
                      c := readkey;
                    end;
                end;
            son := son^.Next;
          end;
      end;
    .
    .
    .
end;
```

Figure 30 : La procédure Transmit modifiée en vue du caching des transtions synchrones.

6.2.3. Résultats

Le Tableau 4compare les temps d'exécution de l'algorithme avec et sans caching et donne également les valeurs de référence qui sont les temps avec et sans caching de l'algorithme DZA&BLE.

	Sched. 4	Sched. 6	Sched 8	Sched 10	Sched 12	Sched 14
DZA&BLE sans caching	0.06	0.28	0.99	6.59	85.3	-
JCC&co sans caching	0.05	0.22	0.98	5.66	26.75	-
DZA&BLE avec caching	0.06	0.11	0.22	0.33	0.44	0.60
JCC&co avec caching	~ 0	0.17	0.88	2.91	10.66	(99.8)

* Tous les temps sont exprimés en secondes.

Tableau 4 : Comparaison de l'algorithme avec caching des transitions synchrones avec les versions connues d'autres algorithmes

On remarque que les temps de l'algorithme avec caching restent exponentiels. Bien que le caching soit efficace (le temps de calcul diminue fortement), cette technique ne permet pas d'obtenir des temps linéaires ou même polynomiaux.

Dans le paragraphe suivant, nous tenterons d'analyser les raisons pour lesquelles le caching n'est pas plus efficace.

6.2.4. Critique de la méthode

Nous allons tout d'abord essayer de donner une explication au manque relatif de performance de notre algorithme avec caching. Notre caching avait pour but de supprimer la redondance dans le calcul des transitions synchrones. Nous pensons avoir fait le maximum dans la mesure de nos possibilités.

Le problème qui subsiste est le suivant. Nous enregistrons des sous-arbres résultats qui sont souvent modifiés lors de l'appel de *Propagate* dans *Transmit*. Cette modification est en général due au fait que plusieurs noeuds sont développés (dont le noeud racine du sous-arbre enregistré) et les transitions effectuées sur ces noeuds entraînent souvent des interactions entre les sous-arbres (par la procédure *AddDescenders* par exemple). Or tout sous-arbre modifié n'est plus d'aucune utilité dans le caching, d'où une certaine perte de performance.

De plus, qu'il y ait caching ou non, l'appel de Propagate est inéluctable. En effet, si on peut enregistrer une opération telle que la création d'un nouveau noeud pour l'exécution d'une transition synchrone, il n'est pas possible d'enregistrer la "propagation de ce noeud", c'est-à-dire l'ensemble des noeuds qui sont créés ou modifiés lors de l'appel de Propagate. En effet, cette propagation fait partie d'un ensemble de propagations et agit sur des composantes de l'arbre spécifique. On est donc obligé de refaire chaque fois le développement des noeuds trouvés ou créés dans la procédure Transmit à l'aide de la procédure Propagate. Ceci représente une perte de temps et une redondance dans les calculs que nous ne sommes pas parvenus à éliminer dans l'optique choisie pour l'algorithme de base. Nous verrons dans la suite qu'il est probablement possible de contourner tous ces problèmes en reconsidérant complètement l'algorithme depuis le début et en choisissant une optique de construction diamétralement opposée.

Un autre problème rencontré lors de l'implémentation est la gestion des noeuds effacés. La technique de stockage des informations dans la table de hachage comporte un inconvénient majeur : il n'est plus possible d'effacer un noeud de l'AP. En effet, la table de hachage contient, entre autres, des pointeurs vers les noeuds "destination" des opérations effectuées. Si, lors des opérations suivantes sur le graphe, on efface ce noeud (en utilisant la procédure DeleteSon par exemple), il y a un risque, par la suite, d'accéder à un noeud effacé (ce qui équivaut à accéder à une cellule mémoire non initialisée). Nous avons donc été obligés de garder en mémoire les noeuds effacés en spécifiant seulement leur suppression par un identificateur -1. Il est évident que cela implique des problèmes de mémoire pour les grosses applications. Nous aurions pu également implémenter une sorte de "garbage collector" qui libérerait de l'espace lorsque celui-ci vient à manquer (par une suppression dans la table de toutes les entrées se servant d'un noeud effacé). Cependant, étant donné les résultats obtenus par l'algorithme sur les exemples cités, il nous a paru préférable de nous tourner vers d'autres voies d'optimisation.

Bien que cette optimisation ne nous ait pas apporté les résultats espérés, elle nous a permis de réaliser que l'algorithme travaillant sur des ASPs au lieu d'APs entraîne également des pertes de performances. En effet, il peut exister à un moment donné deux sous-ASPs identiques dans l'arbre. Ceci implique que malgré le caching implémenté, toutes les opérations se faisant sur l'un des deux arbres ont des chances de devoir être faites sur l'autre aussi. De plus, il n'est pas envisageable de réduire le graphe après chaque opération avec la procédure *STReduce*, car le temps d'exécution en souffre beaucoup trop. Nous n'avions pas, jusque là, envisagé cette autre forme de redondance (également structurelle). Dans l'exposé de l'optimisation suivante, nous montrerons comment nous sommes parvenus à travailler sur un AP réduit et l'impact que cela a eu sur les performances.

6.3. Troisième tentative : Réduction dynamique de l'Arbre Partagé en construction

Dans cette section, nous allons mettre au point une méthode permettant de travailler avec une structure canonique (AP) au lieu d'un arbre semi-partagé.

Nous allons quelque peu rompre l'ordre chronologique dans lequel nous avions annoncé nos optimisations pour plus de facilité de compréhension pour le lecteur. En effet, nous avons implémenté notre technique de deux manières. La première est assez compliquée puisqu'elle modifie l'algorithme en profondeur. La seconde est une simplification de la première beaucoup plus proche de l'algorithme de base et va nous permettre d'introduire l'optimisation réelle de manière plus didactique.

6.3.1. Exposé de la méthode

Pour garder un arbre partagé réduit à tout moment, il faut assurer qu'il n'y ait jamais deux sous-arbres idendiques dans l'arbre en construction. Pour ce faire, il est nécessaire de pouvoir stocker et retrouver chaque sous-AP de l'arbre en construction. Ainsi, on pourra tester, lors de l'ajout d'un fils, si l'arbre résultat de l'opération n'existe pas déjà dans l'AP. Si c'est le cas, on annulera l'opération et on remplacera l'arbre résultat par l'arbre trouvé.

Dans un premier temps, nous exposerons une méthode qui assure qu'au niveau i, tous les sous-APs de niveau j ($j \geq i$) sont uniques. Ensuite nous verrons qu'il y a moyen de garantir l'unicité de tous les sous-APs de l'arbre à tout moment.

6.3.1.1. IDENTIFICATION D'UN SOUS-AP

On peut énoncer une propriété qui définit un sous-AP de manière unique :

Dans un Arbre Partagé AP, tout sous-AP de racine r est identifié par la valeur de r (r^.Info) et par l'ensemble des noeuds fils de r.

En effet, supposons qu'il existe deux noeuds r et r' de même valeur et ayant même ensemble de fils. Alors les sous-arbres de racines r et r' représenteraient le même ensemble d'éléments. En

effet, la première composante de cet ensemble serait la valeur de r et r' qui est identique. En outre, les autres composantes sont des éléments des sous-arbres de racines les fils de r et r'. Or ces fils sont les mêmes.

Cette propriété nous donne un moyen efficace pour retrouver et stocker un sous-AP donné. Il suffit de construire une table de hachage dans laquelle on mettra les informations suivantes :

- *InfSource* : La valeur du noeud racine du sous-AP stocké.
- *Sourceptr* : Pointeur vers le noeud racine du sous-AP stocké.
- *ListSonItem* : Liste de noeuds représentant l'ensemble des fils du noeud Sourceptr.

La fonction de hachage doit être basée sur les informations disponibles lorsqu'on veut insérer ou retirer de l'information de la table. Nous avons à notre disposition la valeur du noeud racine et la liste des fils. Comme la valeur du noeud racine est une valeur beaucoup trop redondante pour une fonction de hachage, nous avons choisi d'implémenter une fonction de l'ensemble des fils.

Cette fonction est la suivante :

Soit *itemarray* le tableau dans lequel les noeuds fils de l'élément recherché ont été insérés et soit *nbr* le nombre de ces fils insérés, alors la position de l'enregistrement possible de ce noeud dans la table *TreeTable* est donné par la formule suivante :

```
if nbr >= 3 then
    mini:=3
  else
    mini:=nbr;
h := ((itemarray[(1 mod mini)+1]^.Ident) div 3 +
      (itemarray[(2 mod mini)+1]^.Ident) div 3 +
      (itemarray[(3 mod mini)+1]^.Ident) div 3) mod MaxTreeTable +1;
```

Bien que la fonction de hachage soit assez lourde dans le sens où elle demande la manipulation de plusieurs éléments pour retrouver l'entrée dans la table, nous disposons avec cette table de hachage d'un outil assez performant pour la réduction dynamique de l'AP en cours de construction.

6.3.1.2. UTILISATION DE LA TABLE DE HACHAGE

La table de hachage définie ci-dessus doit contenir à tout moment la définition de tous les sous-APs de l'arbre en construction. Dans cette optique, toute modification de l'AP (ajout ou suppression d'un noeud, ajout ou suppression d'un fils, ...) devra être précédée d'une recherche dans la table de hachage de manière à éviter d'introduire de la redondance. De plus toute modification effective devra être suivie par une insertion des nouvelles informations dans la table.

Cette méthode appliquée de manière stricte demande des modifications importantes de l'algorithme de base puisqu'il faut traiter chaque endroit où il y a modification de l'AP.

Dans un premier temps, nous allons utiliser un artifice qui nous permettra de garder l'algorithme presque inchangé en introduisant uniquement des traitements supplémentaires en

début et en fin de procédure et en substituant à la procédure *AddDescenders* une procédure *MyUnion*. Nous obtiendrons ainsi une méthode travaillant sur une structure canonique pour tous les niveaux inférieurs au niveau courant. Cet artifice constiste principalement à travailler sur des copies des pères au lieu de travailler sur les pères eux-même. Ceci permet d'effectuer des modifications sans devoir en permanence accéder à la table de hachage. Lorsque l'ensemble des modifications ont été apportées, on consulte alors la table de manière à détecter une possible redondance. Si c'est le cas, la copie est effacée et le vrai père transformé en l'arbre trouvé dans la table. Sinon, le père est assigné à la copie modifiée. Seule la procédure *Propagate* est réellement modifiée. Dans *Transmit*, on a simplement transformé les *AddDescenders* en *MyUnion* pour garantir une structure canonique aux niveaux inférieurs.

Dans un second temps, nous aborderons un technique permettant de garder à tout moment un arbre partagé sans introduction d'artifices. Cette technique est plus intéressante du point de vue des résultats. Cependant, elle utilise des fonctions de modification de l'AP qui transforment l'algorithme de base en un algorithme non strictement top-down (on sera obligé de prendre en compte la partie supérieure de l'AP au niveau courant). En effet, toute modification au niveau courant peut entraîner une non-canonicité des niveaux supérieurs. Il est possible de remonter dans l'arbre partagé à l'aide d'une procédure *Backtracking* et de détecter les redondances aux niveaux supérieurs. L'introduction de cette technique demande également la gestion de nombreux cas différents, ce qui entraîne comme nous le verrons plus tard une modification assez importante du code.

6.3.2. Implémentation

6.3.2.1. LA TABLE DE HACHAGE

On définit la table de hachage de la manière suivante :

```
type
  PTreeRecord = ^TTreeRecord;
  TTreeRecord = record
                  InfSource  : integer;
                  Sourceptr : PItem;
                  ListSonItem : PListItem;
                  Next : PTreeRecord;
                end;
var
  TreeTable : array[1..MaxTreeTable] of PTreeRecord;
```

Le code des procédures de gestion de cette table se trouve en annexe 2.B.1.
Ci-dessous la spécification de ces dernières :

```
Function SearchTable(info,nbr:integer;itemarray:TItemArray) : PItem;
```

Cette fonction recherche dans la table TreeTable une entrée correspondant à un sous-AP dont la valeur de la racine est info et l'ensemble des fils correspond aux noeuds de itemarray.

Paramètres d'entrée :

info : integer. Valeur du noeud racine du sous-AP recherché.
nbr : integer. Nombre de noeuds enregistrés dans *itemarray*.
itemarray : *TItemArray*. Tableau de pointeurs vers des noeuds contenant l'ensemble des noeuds fils de l'AP recherché.

Conditions initiales :

La table *TreeTable* a été initialisée.
Les noeuds de *itemarray* sont triés par ordre croissant de valeur.

Conditions finales :

La fonction renvoie le pointeur vers un noeud (*PItem*) si une entrée de la table contient les valeurs suivantes :

- *InfSource* = *Info*.
- *Sourceptr* = *nodeptr*.
- *ListSonItem* est une liste de noeuds contenant exactement l'ensemble des noeuds de *itemarray*.

La fonction renvoie le pointeur nil dans le cas contraire.

```
procedure RemoveFromTable(x : PItem);
```

Cette procédure supprime l'entrée de la table ayant comme champ "Sourceptr" le paramètre x.

Paramètre d'entrée :

x : PItem. Noeud racine du sous-AP à supprimer de la table.

Conditions initiales :

La table TreeTable a été initialisée.
x <> nil.

Condition finale :

S'il existait une entrée dans la table TreeTable dont le champ *Sourceptr* était égal à x, alors cette entrée a été supprimée de la table.
Sinon, le contenu de la table est inchangé.

```
Procedure InsertInTable(x : PItem);
```

Cette procédure insère dans la table les informations correspondant au sous-AP de racine x.

Paramètre d'entrée :

x : PItem. Noeud racine du sous-AP à insérer dans la table.

Conditions initiales :

La table TreeTable a été initialisée et ne contient pas de sous-AP de racine x.

Conditions finales :

La table TreeTable contient, outre les éléments déjà présents, une entrée comprenant l'information relative au sous-AP de racine x.

6.3.2.2. MY UNION : PROCEDURE DE REMPLACEMENT DE ADDDESCENDERS

Il n'était pas possible de garder la procédure *AddDescenders* telle que nous l'avons vue précédemment. En effet, cette procédure procède à des modifications de l'arbre dans les niveaux inférieurs au niveau courant et devrait par conséquent effectuer tests et insertion dans la table de hachage. De plus, on ne peut modifier un arbre s'il est attaché à plus d'un père.

C'est pourquoi nous avons construit une fonction (*MyUnion*) qui a le même effet que la procédure *AddDescenders* mais qui garde les niveaux inférieurs au niveau courant réduits et qui tient compte des contraintes sus-citées.

Spécification :

```
function MyUnion (a,b : PItem; UtilB : boolean) : PItem;
```

Cette fonction construit un sous-AP correspondant à l'ajout de l'ensemble des descendants de a au sous-AP de racine b.

Paramètres d'entrée :

a, b : *PItem.* Racines des sous-APs origine et destination de l'opération.
UtilB : *boolean.* Booléen qui spécifie si oui ou non on peut réutiliser le noeud *b* lors de l'opération.

Condition initiale :

a et *b* sont deux noeuds de la même couche.
La table *TreeTable* a été initialisée.

Conditions finales :

Le sous-AP de racine *a* est inchangé.
Si *UtilB* = false, alors le sous-AP de racine *b* est inchangé.

La fonction renvoie un sous-AP dont la racine a pour valeur *b^.Info* et dont l'ensemble des descendants est constitué de l'union des descendants de *a* et de *b*. De plus, si un sous-AP équivalent existe dans l'AP, le noeud racine renvoyé est la racine de ce sous-AP.

Algorithme :

La Figure 31 nous montre l'algorithme de la fonction *MyUnion*. La méthode consiste donc à créer un tableau de noeuds contenant l'ensemble des fils de *a* et de *b*. Lorsqu'un fils de *a* a même valeur qu'un fils de *b*, et que ces deux noeuds sont distincts, on appelle la procédure récursivement de manière à effectuer l'union de ces deux sous-arbres.

Lorsque le tableau est rempli, on utilise la fonction de recherche dans la table de hachage pour déterminer s'il existe déjà dans l'arbre un sous-AP équivalent à celui que nous voulons construire (c'est-à-dire ayant même valeur que b et comme fils l'ensemble des fils inséré dans le tableau *Itemarray*).

Si c'est le cas, la fonction renvoie la racine de ce sous-AP. Sinon, soit elle construit un nouveau noeud (*UtilB* = false) de valeur *b^.Info* et dont les fils sont les noeuds de *Itemarray*, soit elle remplace les fils de *b* par l'ensemble des fils trouvés (*UtilB* = true).

```
function MyUnion (a,b : PItem; UtilB : boolean) : PItem;

var
  sonA, sonB : PSon;
  i,nbr : integer;
  result : PItem;

begin
  { Construction d'une liste comprenant l'ensemble des fils de a et de b
    triés par valeur croissante }
  sonA := a^.FirstSon;
  sonB := b^.FirstSon;
  i := 0;
  while (sonA <> nil) or (sonB <> nil) do
    begin
      i := i+1;
      { S'il n'y a plus de fils du côté a alors on les prend du côté b }
      if sonA = nil then
        begin
          itemarray[i] := sonB^.Son;
          sonB := sonB^.Next;
        end
      else
        { S'il n'y a plus de fils côté b, on les prend côté a }
        if sonB = nil then
          begin
            itemarray[i] := sonA^.Son;
            sonA := sonA^.Next;
          end
        else
          { Si la valeur du fils de a est égale à celle du fils de b ...}
          if sonA^.Son^.Info = sonB^.Son^.Info then
            { Si ces noeuds sont identiques on insère ce noeud }
            if sonA^.Son = sonB^.Son then
              begin
                Itemarray[i] := sonA^.Son;
                sonA := sonA^.Next;
```

```
            sonB := sonB^.Next;
          end
        { Si ces noeuds sont distincts, on calcule l'union des deux
          sous-APs dont ils sont racines }
        else
          begin
            result := MyUnion(sonA^.Son,sonB^.Son,false);
            { Insertion du résultat de l'union dans itemarray }
            itemarray[i] := result;
            sonA := sonA^.Next;
            sonB := sonB^.Next;
          end
      { Si la valeur des deux noeuds est différente,
        on insère le plus petit }
      else
        if sonA^.Son^.Info < sonB^.Son^.Info then
          begin
            itemarray[i] := sonA^.Son;
            sonA := sonA^.Next;
          end
        else
          begin
            itemarray[i] := sonB^.Son;
            sonB := sonB^.Next;
          end;
  end;
{ Recherche d'un sous-AP correspondant à b^.Info et à la liste créée }
nbr := i;
result := SearchTable(b^.Info,nbr,itemarray);
{ Si aucun sous-AP n'est trouvé, on construit le nouveau noeud ou on
  modifie le noeud b (en fonction de UtilB) et on met a jour la table}
if result = nil then
  begin
    if UtilB then
      begin
        RemoveFromTable(b);
        sonB := b^.FirstSon;
        i:=1;
        while i <= nbr do
          begin
            if sonB <> nil then
              if itemarray[i]^.Info = sonB^.Son^.Info then
                begin
                  if itemarray[i] <> sonB^.Son then
                    ReplaceSon(b,sonB^.Son,itemarray[i]);
                  sonB := sonB^.Next;
                end
              else
                AddSon(b,itemarray[i])
            else
              AddSon(b,itemarray[i]);
            i := i+1;
          end;
        result := b;
        InsertInTable(result);
      end
    else
      begin
        layerptr := GetLayer(PSync2,niv);
        result := AddItem(layerptr,b^.Info);
        i := 1;
        while i<= nbr do
          AddSon(result,itemarray[i]);
```

```
          InsertInTable(result);
        end;
    end;
  { La fonction renvoie le résultat trouvé ou construit }
  MyUnion := result;
end;
```

Figure 31 : Algorithme de la fonction MyUnion.

6.3.2.3. MODIFICATION DE LA PROCEDURE PROPAGATE

Nous avons modifié la procédure *Propagate* en remplaçant dans un premier temps l'appel à la procédure *AddDescenders* par un appel à la fonction *MyUnion* suivi d'une gestion du résultat pour l'insertion dans l'arbre des modifications.

Nous avons également décidé de travailler sur des copies. Ainsi, au début de la procédure *Propagate*, on crée un noeud qui est la réplique du noeud *father* (à l'identifiant près). Ceci permet de ne pas considérer les modifications qu'il peut y avoir aux niveaux supérieurs puisque ce noeud copie n'est pas relié à ces niveaux. A la fin de la procédure, une recherche est effectuée de manière à déceler s'il existe dans l'AP un sous-AP équivalent à celui de racine le noeud copie. Si un tel sous-AP est trouvé, alors la racine de celui-ci est passée comme paramètre résultat de la procédure (c'est-à-dire *father* est assigné à ce noeud). Dans le cas contraire, on ajoute à *father* les descendants de *copfather* et on met à jour la table de hachage.

La Figure 32 nous montre le début et la fin de la procédure *Propagate* modifiée. Pour une version plus complète de la procédure, le lecteur se référera à l'annexe 2.B.2.

```
Procedure Propagate(var slistptr:PIntList;var father:PItem);

 var
   copfather : PItem;
   .
   .
   .
 begin
   { Création d'une copie du noeud father, sauf si ce noeud est la racine
     de l'arbre root (dans ce cas aucune copie n'est nécessaire) }
   if niv > 1 then
     begin
       layerptr := GetLayer(PSync2,niv-1);
       copfather := AddItem(layerptr,father^.Info);
       sonptr := father^.FirstSon;
       while sonptr <> nil do
         begin
           AddSon(copfather,sonptr^.Son);
           sonptr := sonptr^.Next;
         end;
     end
   { Si father est la racine de l'arbre, on assigne la copie à l'original }
   else
     copfather := father;

   while slistptr <> nil do
     begin
```

```
      { Corps de la procédure modifié en introduisant la fonction
        MyUnion et la gestion des résultats de celle-ci }
      .
      .
      .
    end;

  { Construction d'une liste contenant l'ensemble des fils de la copie }
  sonptr := copfather^.FirstSon;
  nb := 0;
  while sonptr <> nil do
    begin
      nb := nb+1;
      itemarray[nb] := sonptr^.Son;
      sonptr := sonptr^.Next;
    end;
  { Recherche d'un sous-AP équivalent à celui
    de racine la copie dans l'AP}
  newfather := SearchTable(copfather^.Info,nb,itemarray);
  { Si aucun arbre n'a été trouvé,
    alors on assigne le résultat à la copie,
    ce qui équivaut à ajouter à father les descendants de copfather }
  if newfather = nil then
    begin
      RemoveFromTable(father);
      AddDescenders(copfather,father);
      InsertInTable(father);
      if (father <> copfather) and (niv > 1) then
        begin
          layerptr := GetLayer(PSync2,niv-1);
          DeleteItem(layerptr,copfather);
        end;
    end
  { Si un sous-AP a été trouvé et si ce sous-AP n'est pas la copie
    elle-même, on supprime la copie (si ce n'est pas root) et on
    assigne father au noeud trouvé }
  else
    begin
      if niv > 1 then
        begin
          if newfather <> copfather then
            begin
              layerptr := GetLayer(PSync2,niv-1);
              DeleteItem(layerptr,copfather);
            end;
        end;
      if father <> newfather then
        begin
          father:=newfather;
        end;
    end;
end;
```

Figure 32 : Une partie des modifications apportées à la procédure Propagate

6.3.2.4. MODIFICATIONS APPORTEES A LA PROCEDURE TRANSMIT

Nous avons modifié la prodédure *Transmit* pour y inclure la procédure *MyUnion*. Cette modification est identique à celle effectuée dans *Propagate* et nous n'y reviendrons donc pas.

En outre, nous avions rencontré lors des tentatives précédentes des problèmes d'overflow dus au fait que l'on créait beaucoup de noeuds inutiles. Comme les identifiants sont des entiers, on ne pouvait créer plus de MAXINT noeuds, même ci ceux-ci étaient supprimés par après. C'est pourquoi nous avons décidé de "travailler à l'économie". On ne créera donc un noeud que si cela s'avère indispensable. En particulier, on ne crée plus le noeud *res* avant l'appel de *Transmit* mais celui-ci devient un paramètre résultat créé si nécessaire lors de l'exécution de *Transmit*.

Ce changement entraîne inévitablement une surcharge du code et il n'est pas spécialement intéressant de détailler les modifications que cela a imposés. Le lecteur intéressé pourra trouver tous les détails d'implémentation en annexe 2.B.3.

La Figure 33 nous montre une partie de la procédure *Transmit* où l'on peut voir l'utilisation que l'on fait de la fonction *MyUnion*, la gestion de la table de hachage au niveau courant et les problème de gestion du paramètre *res*.

```
{ Calcul de l'union de son^.Son avec nextpitem en réutilisant nextpitem }
newson := myunion(son^.Son,nextpitem,true)
newson^.First := true;
{ Si le résultat n'est pas nextpitem (un arbre équivalent a été trouvé)}
if newson <> nextpitem then
  begin
    { Remplacer le fils nextpitem de res par newson }
    ReplaceSon(res,nextpitem,newson);
    { Si nextpitem n'est plus rattaché au niveau supérieur, le supprimer}
    if nextpitem^.Nbfathers = 0 then
      begin
        RemoveFromTable(nextpitem);
        layerptr := Getlayer(PSync2,syncniv);
        DeleteItem(layerptr,nextpitem);
      end;
    { Ajouter newson à l'ensemble des noeuds à développer }
    if not newson^.ToBeDev then
      begin
        newson^.ToBeDev := TRUE;
        AddListItem(slistptr,next);
      end;
  end;
```

Figure 33 : Exemple de l'utilisation de la table de hachage et de la fonction MyUnion.

6.3.3. Modification de l'implémentation

Dans ce paragraphe, nous allons exposer la manière dont nous sommes parvenus à travailler sur un AP réduit à tout moment, tout en évitant des artifices de calcul tels que la copie utilisée dans *Propagate*.

Le problème auquel nous avons été confronté était le suivant : lorsqu'on modifie un sous-arbre à un niveau donné ou, plus précisément, lorsqu'on remplace un noeud fils par un autre noeud trouvé dans la table de hachage, les modifications apportées à l'AP ont un impact sur tout le

graphe et peuvent introduire de la redondance non seulement dans les niveaux inférieurs (ce qui est géré par la première méthode), mais également dans les niveaux supérieurs.

Dans le paragraphe précédent, nous avions contourné le problème en introduisant la notion de copie et en testant à la sortie de *Propagate* si la copie ne correspondait pas à un sous-AP. Cela permettait de ne pas prendre en compte les niveaux supérieurs, la réduction étant faite chaque fois que l'on remontait d'un niveau.

Dans ce paragraphe, nous allons prendre en compte explicitement les niveaux supérieurs. Nous espérons ainsi, par une méthode de réduction plus dynamique (on n'a plus seulement réduction à la fin de *Propagate*), obtenir de meilleurs résultats.

Les niveaux supérieurs doivent être pris en compte lorsqu'une substitution est effectuée au niveau courant, c'est-à-dire lorsqu'un noeud fils est remplacé par un autre noeud. En effet, il faut alors vérifier que l'ensemble de fils ainsi créé, associé à la valeur du noeud père, ne constitue pas déjà un sous-AP de l'arbre en construction. Si c'est le cas, le noeud père devra être remplacé par le noeud racine du sous-AP trouvé et la procédure devra être réitérée au niveau supérieur et cela pour tous les "grand-pères" ayant le père donné comme fils.

La Figure 34 nous montre un exemple de substitution d'un noeud qui entraîne des modifications dans les niveaux supérieurs. L'appel de *Backtracking* s'effectue après que l'on ait tenté d'effectuer la transition $b \xrightarrow{\tau} d$.

Une recherche dans la table nous informe qu'il existe déjà un noeud c ayant l'ensemble pour ensemble de fils les noeuds *a*, *b*, *d*. La procédure *BackTracking* est donc appelée pour l'ancien père (le *c* barré) et le nouveau (le *c* trouvé), puisque le noeud *c* barré doit être remplacé par le nouveau *c* (comme indiqué sur le schéma).

Dans la procédure *BackTracking*, on examine tous les pères de *c* barré (ce sont les noeuds *a* et *b*). La recherche dans la table nous fournit un nouveau *a* (noeud de valeur a ayant nouveau *c* comme fils). Par contre, il n'existe pas encore de noeud de valeur *b* ayant nouveau *c* comme fils. On va donc appeler la procédure *BackTracking* récursivement pour *a*, puisque l'ancien *a* doit être remplacé par le nouveau (comme indiqué sur le schéma).

En ce qui concerne *b*, on va procéder au remplacement du fils ancien *c* par le fils nouveau *c* (arc marqué "supprimé" et "ajouté") et on peut sortir de la procédure. L'appel de *BackTracking* pour ancien *a* et nouveau *a* a pour résultat le remplacement du fils ancien *a* de *sj* par le fils nouveau *a* (arc marqué "supprimé" et "ajouté").

De plus ces opérations entraînent que certains noeuds se retrouvent sans père. Dans ce cas, ces noeuds sont supprimés, ainsi que leurs entrées dans la table (noeuds barrés sur le schéma).

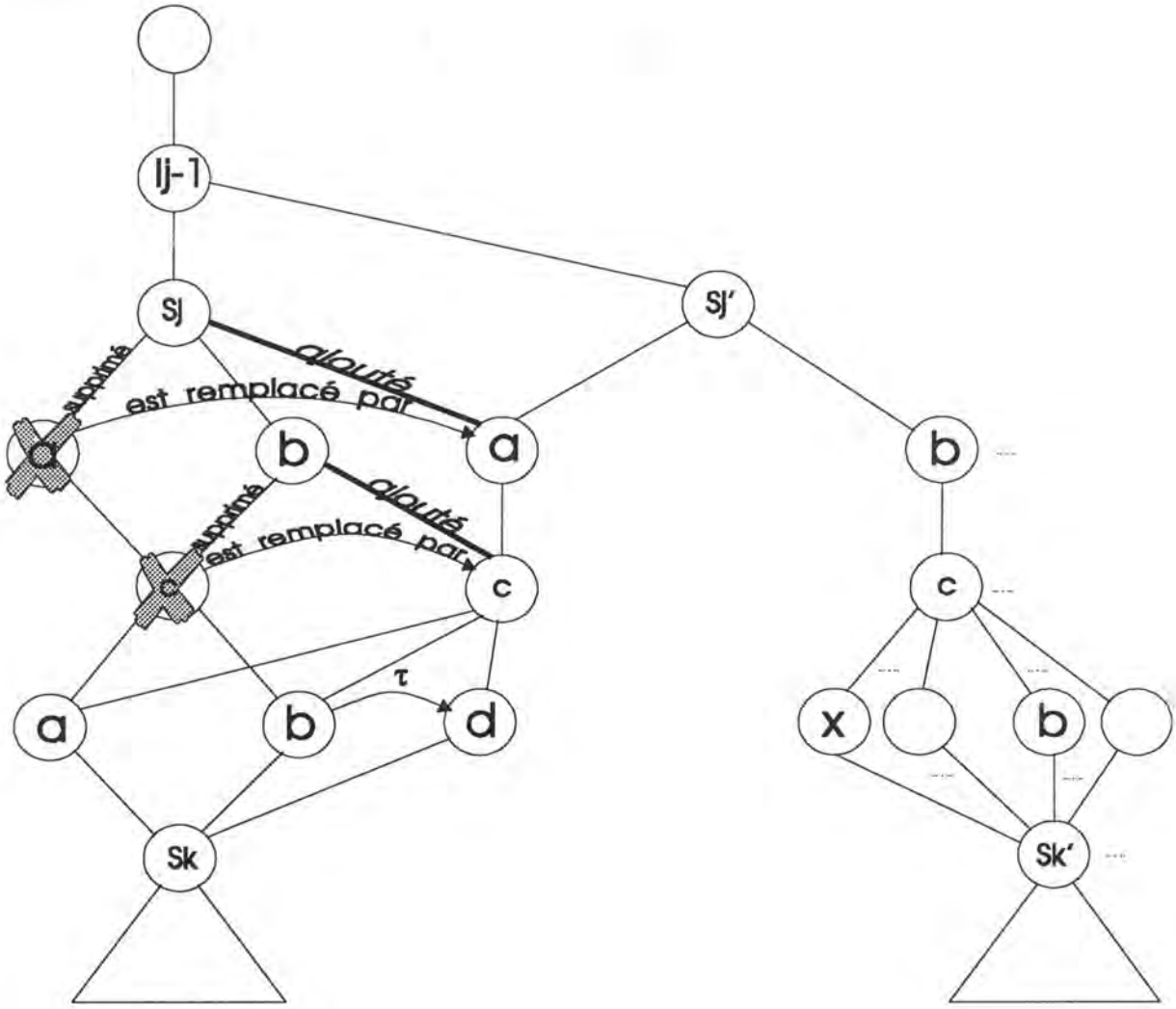

Figure 34 : Schéma d'exécution de la procédure BackTracking

La procédure qui effectue l'examen des niveaux supérieurs, lorsque c'est nécessaire, est appelée *BackTracking* (voir Figure 35). Cette procédure récursive parcourt le niveau supérieur en recherchant l'ensemble des noeuds ayant *oldit* comme fils. Pour chaque père, on recherche dans la table de hachage un arbre équivalent à celui qu'on obtiendrait en remplaçant le fils *oldit* par le fils *newit*. Si un sous-AP est trouvé, on appelle *BackTracking* récursivement avec le père examiné et la racine (nouveau père) du sous-AP trouvé dans la table. Sinon, on effectue le remplacement voulu dans l'ensemble des fils de père (on remplace oldit par newit).

```
procedure BackTracking (var oldit,newit : PItem);

var
  maxfather,nb,nbr,i : integer;
  curfather,newfather,item : PItem;
  sonptr : PSon;
```

```
begin
  { On va examiner le niveau du comprenant les pères de oldit }
  niv := niv - 1;
  layerptr := GetLayer(PSync2,niv);
  { Boucle sur l'ensemble des pères jusqu'à ce que tous les pères
    de oldit aient été examinés }
  maxfather := oldit^.NbFathers;
  curfather := layerptr^.FirstItem;
  nb := 0;
  while nb < maxfather do
    begin
      { Si le père courant possède oldit comme fils,...}
      item := HasSon(curfather,oldit^.info);
      if item = oldit then
        begin
          nb := nb+1;
          i := 0;
          { Construction de la liste des fils de curfather en
            remplaçant oldit par newit }
          sonptr := curfather^.FirstSon;
          while sonptr <> nil do
            begin
              if sonptr^.Son = oldit then
                begin
                  i := i+1;
                  itar[i] := newit;
                end
              else
                begin
                  i := i+1;
                  itar[i] := sonptr^.Son;
                end;
              sonptr := sonptr^.Next;
            end;
          nbr := i;
          { Recherche d'un sous-AP équivalent dans la table de hachage }
          newfather := SearchTable(curfather^.Info,nbr,itar);
          { Si un nouvel arbre est trouvé, alors effectuer le
            backtracking sur le père courant et le nouveau père }
          if newfather <> nil then
            backtracking(curfather,newfather)
          { Sinon, remplacer oldit par newit dans l'ensemble des fils
            de curfather et mettre la table de hachage à jour}
          else
            begin
              RemoveFromTable(curfather);
              replaceson(curfather,oldit,newit);
              InsertInTable(curfather);
              { Si oldit ne possède plus de fils, alors le supprimer }
              if oldit^.NbFathers = 0 then
                begin
                  RemoveFromTable(oldit);
                  layerptr := GetLayer(PSync2,niv+1);
                  DeleteItem(layerptr,oldit);
                end;
            end;
          { Passer au père suivant }
          curfather := curfather^.Next;
        end
      { Si le père courant ne possède pas oldit comme fils,
        passer au père suivant }
      else
        curfather := curfather^.Next;
```

```
    end;
  { Remettre la variable globale niv à sa valeur initiale }
  niv := niv+1;
end;
```

Figure 35 : Algorithme de la procédure BackTracking

Cette procédure est appelée chaque fois qu'un remplacement de fils (ReplaceSon) est effectué, si ce remplacement peut entraîner des redondances au niveau supérieur. On peut ainsi garantir à tout moment un graphe réduit sans utiliser d'artifice de calcul.

Cette manière de procéder a cependant plusieurs inconvénients. Tout d'abord elle alourdit considérablement le code des procédures *Propagate* et *Transmit* étant donné le nombre de cas différents à gérer chaque fois que l'on exécute une transition (pour plus de détails, se référer au code commenté des procédures *Propagate* et *Transmit* exposé en annexe 2.C.1 et 2.C.2). De plus, elle requiert de parcourir des niveaux entiers pour la recherche des pères (vu la structure de données orientée top-down). Nous avons envisagé de construire une table donnant l'ensemble des pères de chaque fils, mais cela aurait constitué une complication supplémentaire du code sans garantir pour cela une amélioration conséquente des résultats.

6.3.4. Résultats

Le Tableau 5 compare les temps d'exécution de l'algorithme travaillant sur un arbre réduit, première version et deuxième version, avec les valeurs de référence qui sont les temps avec et sans caching de l'algorithme DZA&BLE, ainsi que les temps de l'algorithme de base.

Schedulers	4	6	8	10	12	14	16	18
DZA&BLE sans caching	0.06	0.28	0.99	6.59	85.3	-	-	-
JCC&co alg. de base	0.05	0.22	0.98	5.66	26.75	-	-	-
JCC&co réduit simple	~ 0	0.11	0.44	3.8	31.00	619	-	-
JCC&co réduit + backtracking	0.05	0.17	0.60	2.48	10.00	41.96	166.3	735.5
DZA&BLE avec caching	0.06	0.11	0.22	0.33	0.44	0.60	0.83	1.06

Tableau 5 : Comparaison des algorithmes travaillant sur un arbre réduit avec les autres.

Les temps obtenus sont assez satisfaisants en tout cas en ce qui concerne l'algorithme avec backtracking. Cependant, ils restent exponentiels. On peut remarquer que l'algorithme avec backtracking est beaucoup plus performant que l'algorithme réduit simple. Nous verrons dans le paragraphe suivant comment on peut expliquer cela.

A la lumière de ces chiffres, et vu les optimisations que nous avons déjà tentées, il nous apparaît que l'algorithme tel qu'il a été implémenté semble ne pas pouvoir rivaliser avec son homologue.

Toutefois, nous exposerons dans la section suivante, et de manière succinte, l'optimisation de la dernière chance que nous avons réalisée et qui consiste à greffer sur le dernier alogorithme (JCC&co avec backtracking) la technique d'optimisation exposée dans la section précédente (le caching des transitions synchrones).

6.3.5. Critique et comparaison des deux méthodes

Tout d'abord, on peut constater que le premier algorithme exposé, c'est-à-dire celui travaillant sur un arbre réduit pour les niveaux inférieurs au niveau courant n'est pas beaucoup plus performant que l'algorithme de base.

Le fait qu'on travaille sur des copies d'une part et qu'on ne tienne pas compte des niveaux supérieurs, d'autre part, permet d'expliquer cela. En fait, la technique exposée est quasi-équivalente à effectuer une réduction du graphe à la fin de *Propagate*. L'avantage de la méthode est que cette réduction est faite de manière extrêmement rapide (elle ne nécessite pas le parcours de tous les noeuds du graphe grâce à la table de hachage) mais elle n'est pas fondamentalement différente de l'algorithme de base.

Par contre, dès que l'on tient compte des niveaux supérieurs, les temps de calculs sont nettement meilleurs. En effet, lorsqu'une transition est effectuée, elle n'est plus seulement effectuée par rapport à un père donné, mais on propage l'effet de cette transition à tous les "ancêtres" qui peuvent être concernés par celle-ci grâce à la procédure *BackTracking*. La transition en question ne sera effectuée qu'une fois au lieu d'être effectuée chaque fois que l'on arrive à ce même noeud par un autre chemin. C'est cela qui fait la grande différence dans les temps de calcul des deux algorithmes.

Pourquoi l'algorithme avec BackTracking n'est-il pas encore plus performant ? Toute l'optimisation effectuée est basée sur la structure. Cependant, la redondance au niveau du calcul des transitions est encore présente. En effet, lors du calcul d'une transition, il n'y a pas de caching. Le résultat de la transition est calculé et c'est seulement à ce moment qu'il est possible d'aller voir si un résultat équivalent se trouvait déjà dans l'arbre. On n'a donc pas un caching au niveau des transitions mais bien au niveau de la structure.

De plus, la table de hachage est basée sur une liste de noeuds (seul moyen de ne pas introduire de la redondance). Le temps d'accès à cette table n'est donc plus négligeable et les temps de gestion interviennent pour une plus grande part dans les temps de calcul de l'algorithme.

Dans la section suivante, nous verrons d'ailleurs que l'association des optimisations que nous allons effectuer entraîne des temps de gestion probablement trop importants pour fournir des résultats vraiment meilleurs.

Quoi qu'il en soit, les algorithmes mis à jour dans cette section constituent un résultat en soi, puisqu'ils sont les premiers algorithmes de calcul des états accessibles travaillant sur une structure canonique à tout moment. Cette structure garantit entre autre un espace minimal en mémoire, ce qui n'est pas négligeable lorsqu'on travaille sur de gros systèmes. Les techniques

utilisées pourraient donc inspirer des optimisations d'algorithmes plus performants en vue de travailler sur des structures réduites requérant un espace mémoire minimum.

6.4. Ultime tentative : Caching des transitions syncrhones sur l'algorithme travaillant sur la structure canonique.

6.4.1. Exposé de la méthode

Dans les sections précédentes, nous avons envisagé deux optimisations possibles qui se sont révélées plus ou moins efficaces : Le caching des transitions synchrones et la réduction dynamique de l'AP en construction.

Ces deux optimisations sont complètement différentes et on peut donc se demander si les performances ne pourrait pas être encore bien meilleures en utilisant ces deux optimisations en même temps. On travaillerait donc alors sur un graphe réduit dynamiquement tout en effectuant un caching des transitions synchrones.

Nous n'exposerons pas dans les détails la manière dont cette dernière optimisation a été réalisée. Disons simplement que l'optimisation par réduction dynamique de l'arbre avait été implémentée de manière à pouvoir facilement greffer le caching des transitions synchrones dessus. Moyennant quelques modifications, nous sommes parvenus sans trop de difficultés à associer ces deux techniques.

6.4.2. Implémentation

cfr. code des procédures optimisée (Annexe 2.C.1 et 2.C.2)

6.4.3. Les résultats

Le Tableau 6 nous donne un aperçu des résultats obtenus. Ils sont comparés aux temps obtenus lorsqu'on utilise les deux optimisations séparément. On constate que cette dernière optimisation a permis de diminuer les temps de calcul de l'algorithme avec réduction dynamique d'un facteur 2.5.

Schedulers	4	6	8	10	12	14	16	18
JCC&co alg. de base	0.05	0.22	0.98	5.66	26.75	-	-	-
JCC&co Cache trans. sync.	~ 0	0.17	0.88	2.91	10.66	(99.8)	-	-
JCC&co réduit + backtracking	0.05	0.17	0.60	2.48	10.00	41.96	166.3	735.5
JCC&co Opti. combinées	0.05	0.05	0.28	0.99	3.68	14.56	55.91	237.0

* Tous les temps sont exprimés en secondes.

Tableau 6 : Optimisations combinées vs optimisations séparées.

6.4.4. Critique de la méthode

Au vu des résultats que cette optimisation avait eu sur l'algorithme de base, on pourrait s'étonner que ce facteur soit si peu élevé. Cependant, si l'on se souvient de la technique utilisée pour cette optimisation, on remarque que le caching se fera moins souvent si l'arbre est réduit à tout moment. En effet, une information enregistrée n'est considérée comme valide que si les sous-arbres impliqués n'ont pas été modifiés depuis l'enregistrement. Or, par la structure même de l'arbre partagé, l'espace de stockage est minimum, et par conséquent, les modifications dans cet espace sont assez fréquentes. Donc, statistiquement, le caching échouera plus lorsqu'on travaille sur un AP que losrqu'on utilise un ASP.

Quoi qu'il en soit, les temps restent exponentiels et il est plus que probable qu'aucune optimisation ne puisse améliorer beaucoup plus les temps de calcul. Les différentes analyses que nous avons pu faire en réalisant ces implémentations nous ont amenés à penser que peut-être l'optique dans laquelle nous avons implémenté notre algorithme n'était pas la meilleure. Nous verrons dans un dernier paragraphe une autre manière de concevoir la construction de l'arbre partagé qui, bien qu'étant a priori moins naturelle, pourrait après réflexion s'avérer plus efficace, en tout cas en ce qui concerne les temps de calcul.

7. En conclusion : une optique d'implémentation différente

Dans l'exposé, nous avons toujours considéré la construction de l'AP contenant l'ensemble des états accessibles comme la modification progressive d'un AP initial ne contenant que l'état initial de l'automate produit.

Nous avons également constaté que le fait de modifier la structure empêchait l'utilisation efficace d'un caching sur les transitions. Il nous apparaît après cette étude que sans caching sur les transitions, il n'est probablement pas possible de réduire les temps de calcul à des temps linéaires ou même polynomiaux...

Cependant, il est possible de construire l'AP sans modifier l'arbre de départ. On entend par non-modification le fait de ne jamais modifier un sous-arbre de l'AP en construction. Ainsi, on pourrait implémenter une technique travaillant sur un arbre réduit dynamiquement comme vu précédemment et pour lequel chaque sous-AP créé ne serait jamais modifié. Lorsqu'en temps normal on aurait modifié l'arbre, on créerait un nouvel arbre contenant les modifications et on propagerait l'opération en utilisant une procédure du même style que *Backtracking*.

Cette technique tout à fait différente permettrait un caching des transitions très efficace. En effet, on pourrait garantir que toute opération faite sur un sous-AP a été enregistrée et ne devra plus jamais être recalculée. On pourrait même implémenter un caching non pas des transitions, mais de l'ensemble des transitions associées à un noeud donné. Ainsi, losrqu'on devrait à nouveau développer le même sous-AP, on aurait immédiatement le résultat de l'application de toutes les transitions possibles à ce sous-AP. Il est certain qu'une telle technique serait beaucoup plus efficace que toutes celles que nous avons envisagées dans ce mémoire.

Cependant, comme le dit l'adage, on ne fait pas d'omelette sans casser des oeufs. En effet, la construction d'un arbre de cette manière entraînerait probablement un besoin en mémoire assez important. Sans pouvoir le chiffrer a priori, on remarquera toutefois que cette technique implique que l'AP en construction devra être recopié au moins partiellement chaque fois qu'une transition est effectuée. En effet, aucun AP, y compris ceux dont les racines sont les "ancêtres" du noeud courant, ne peut être modifié. Ceci implique une copie de tous les "ancêtres" d'un père, chaque fois que le sous-AP de racine ce père devrait être modifié.

On pourrait sans doute se servir d'une technique de garbage collecting éliminant les versions les plus anciennes et non incluses dans l'AP courant de la mémoire et des différentes tables de hachage. Il faut cependant se demander si ce garbage collecting serait si facile à implémenter, étant donné qu'il serait nécessaire de tester qu'un sous-AP ne fait plus partie de l'AP en construction avant de l'éliminer.

Quoi qu'il en soit, ceci constitue une voie d'avenir qui vaudrait la peine d'être explorée. Le temps nous a manqué mais nous ne doutons pas que quelqu'un soit intéressé à la réalisation de ce projet.

Chapitre 4
Récapitulatifs et comparaison des algorithmes

Dans ce chapitre, nous faisons un récapitulatif de tous les résultats obtenus dans ce mémoire. Nous distinguons deux aspects fondamentaux dans la recherche des états accessibles du produit synchronisé d'un système d'automates : D'une part le temps de calcul qui est primordial dans cette matière, d'autre part, l'espace occupé qui, bien qu'étant souvent négligé, limite parfois certains algorithmes trop gourmands.

1. Les temps de calcul

Nous avons exposé dans ce mémoire l'algorithme de base et les différentes optimisations que nous avons faites. Nous allons à présent faire un récapitulatif de tous les résultats obtenus de manière à pouvoir situer notre algorithme par rapport à tous ceux qui ont été réalisés précédemment.

Le Tableau 7 suivant nous donne un aperçu des temps des algorithmes considérés appliqués aux Schedulers de Milner.

# automates	6	8	10	12	14	16	18	20
# états	577	3073	15361	73729	$34\ 10^4$	$15\ 10^5$	$7\ 10^6$	$31\ 10^6$
Fernandez	2.6	21	160	-	-	-	-	-
Groote	0.2	1.2	7.4	53	-	-	-	-
Enders	21	40	87	145	233	348	569	850
Rauzy	0.7	1.3	2.03	3.06	4.41	5.76	7.36	9.36
DZA&BLE	0.28	0.99	6.59	85.3	-	-	-	-
DZA&BLE + cache	0.11	0.22	0.33	0.44	0.60	0.83	1.06	1.31
JCC&co Algorithme de base	0.22	0.98	5.66	26.75	-	-	-	-
JCC&co Cache trans. sync.	0.17	0.88	2.91	10.66	(99.8)	-	-	-
JCC&co Réduction simple	0.11	0.44	3.8	31.00	619	-	-	-
JCC&co Réduction$^+$	0.17	0.60	2.48	10.00	41.96	166.3	735.5	-
JCC&co Cache+Réduction$^+$	0.05	0.28	0.99	3.68	14.56	55.91	237.0	-

* Tous les temps sont exprimés en secondes.

Tableau 7 : Tableau récapitulatif des résultats obtenus.

Sur les graphes suivants, il est plus facile de situer les différentes versions de notre algorithme et de constater l'effet des différentes optimisations sur les temps de calculs. Le premier nous donne l'évolution de notre algorithme avec les différentes optimisations et est comparé à DZA&BLE sans cache. Le second situe notre algorithme le plus performant dans le panel présenté ci-dessus.

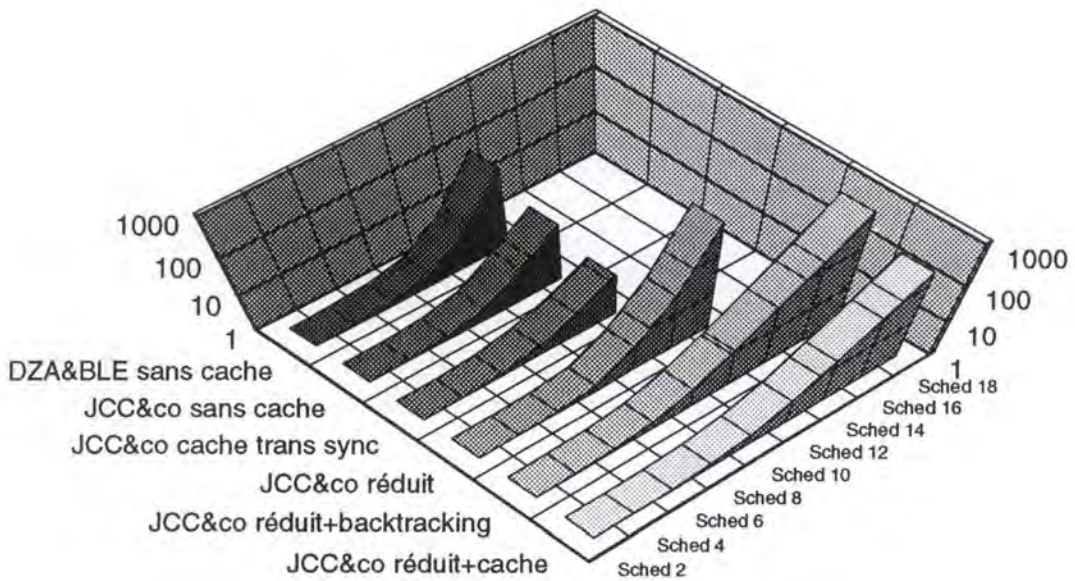

Figure 36 : Evolutions des temps d'exécution de l'algorithme mis au point (échelle logarithmique)

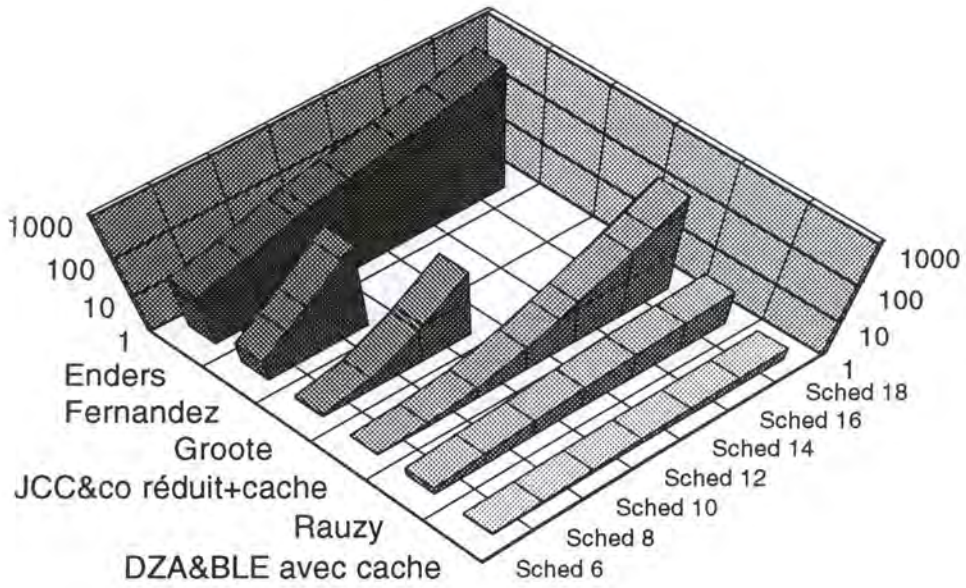

Figure 37 : Comparaison des meilleurs temps d'exécution des algorithmes existants (échelle logarithmique)

On remarque sur le premier graphique que l'évolution des performances de notre algorithme est assez bonne. D'une part, les optimisations ont permis de traiter des exemples plus important en taille, d'autre part, les temps d'exécution se sont nettement améliorés.

On peut voir sur les graphes, qui ont une échelle des temps logarithmiques, que la courbe de l'algorithme de base est encore une exponentielle, malgré cette échelle. Il en est de même pour toutes les optimisations qui ne considèrent que les niveaux inférieurs au niveau courant.

Par contre, dès qu'on prend en compte tous les niveaux, grâce à la procédure BackTracking vue dans le chapitre précédent, on obtient pour ainsi dire une exponentielle pure, dès que les coûts de gestion deviennent négligeable par rapport aux temps de l'algorithme. A partir du Scheduler 8, la courbe est pratiquement droite, ce qui correspond à une exponentielle pure puisqu'on à une échelle logarithmique.

Il est difficile de tirer des conclusions sûres à partir de telles observations. On peut avancer que le fait de ne tenir compte que des niveaux inférieurs comme il était prévu dans l'algorithme de base ne permet pas d'obtenir des temps raisonnables. En effet, on est alors obligé de faire plusieurs fois les mêmes opérations. Comme, dans l'optique d'implémentation choisie, le caching des opérations est impossible, il est indispensable de minimiser les redondances dans l'exécution des transitions.

On remarque toutefois que, malgré l'échec dans la recherche d'un algorithme linéaire, notre algorithme a des temps comparables et même meilleurs que la plupart des algorithmes existants. Seul la méthode de Rauzy, linéaire elle-même permet l'obtention de meilleures performances. Notons toutefois que la méthode de Enders, quoiqu'étant mal classée, est cependant meilleure que la nôtre puisqu'elle n'est pas exponentielle. Il est probable que pour des exemples plus conséquents les temps d'exécution de Enders soient meilleurs que les nôtres.

2. L'espace occupé

L'espace occupé au cours de l'exécution de l'algorithme constitue également un point important dans l'évaluation d'un algorithme. Dans notre cas, celui-ci a constitué une limite lorsqu'on utilisait la méthode de caching des transitions. En effet, nous avons vu qu'il était alors difficile de récupérer l'espace mémoire lorsqu'on supprimait des noeuds.

Ce problème nous a obligé à ajouter une table permettant de déterminer si un noeud donné avait été supprimé. Ceci nous a permis de réutiliser l'espace occupé et ainsi de présenter les temps d'exécution de la méthode pour le Scheduler 14, malgré l'handicap que constituait l'ajout de nouveaux tests et de nouvelles tables.

En ce qui concerne les algorithmes travaillant sur arbre réduit, il est certain qu'ils constituent une des méthodes les moins coûteuses en espace mémoire. La table de hachage créée contient autant d'entrées qu'il y a de noeuds dans le graphe (ce qui est un minimum), et l'arbre construit est à tout moment minimal. Nous avons ainsi obtenu un algorithme travaillant sur une structure minimale du point de vue espace mémoire.

Les algorithmes travaillant sur un arbre semi-partagé qu'ils réduisent régulièrement (chaque itération pour DZA&BLE ou après Transmit pour JCC&co) créent des structures plus vastes qu'ils réduisent au minimum à chaque réduction. On peut avancer que, pour les plus gros exemples, ces réductions devraient se faire plus souvent, si l'on ne dispose pas de mémoire suffisante. De ce point de vue, notre algorithme deviendrait plus compétitif.

Dès que l'on introduit le caching des transitions synchrones, on a de nouveau le même problème qu'auparavant (aucun noeud ne peut être réellement supprimé). Cependant, cet inconvénient n'est pas très marqué, étant donné la technique adoptée consistant à créer un noeud uniquement lorsqu'on est sûr qu'il n'y a pas moyen de faire autrement. Ceci permet à l'algorithme de fonctionner au-delà des 18 schedulers.

Conclusion

Durant ce mémoire, nous avons développé un nouvel algorithme de calcul des états accessibles. L'algorithme de base que nous avons implémenté présente des résultats équivalents à ceux de DZA&BLE sans caching. Ceci nous a amené à penser que nous pourrions, par des optimisations efficaces, concurrencer l'algorithme DZA&BLE avec cache.

Les diverses optimisations effectuées ont permis d'améliorer les temps de calcul de notre algorithme de manière substantielle. De plus, ces optimisations ont permis d'envisager des cas plus importants en taille et en nombre d'automates synchrones.

Dans un premier temps, nous nous sommes contentés de travailler sur une structure d'arbre semi-partagé comme le fait DZA&BLE. Cependant, les résultats obtenus n'étaient pas convaincants et une analyse profonde des exécutions des différentes versions mises au point nous ont permis de déceler les redondances dans les opérations effectuées dues à la structure même utilisée.

C'est pourquoi nous avons envisagé d'implémenter une nouvelle version de l'algorithme travaillant sur un arbre partagé canonique. Lors de cette implémentation, nous avons pu voir que la prise en compte de tous les niveaux de l'arbre construit était essentielle pour de meilleures performances (implémentation de la méthode de backtracking). Cependant, le "backtracking" a engendré, à notre plus grand regret, une perte d'identité de notre algorithme qui était originellement orienté "top-down".

L'ensemble des optimisations réalisées et combinées nous offrent un algorithme aux temps concurrentiels puisqu'il se situe parmi les meilleurs algorithmes existant. Nous devons toutefois déplorer que tous les efforts fournis et toutes les possibilités d'optimisation envisagées ne nous aient pas permis d'obtenir des perfomances équivalentes à l'algorithme DZA&BLE.

Le résultat le plus probant est sans aucun doute d'avoir mis au point le premier algorithme de calcul des états accessibles travaillant sur la structure d'arbre partagé à tout moment canonique. Ce résultat montre qu'il est possible de travailler sur une structure canonique, d'une part, et d'autre part, que les temps de calcul sur une telle structure sont meilleurs. En effet, on peut noter la nette amélioration entre les temps de calcul sur les arbres semi-partagés et les temps de calcul finaux. Ceci nous incite à penser qu'il doit être possible de construire un algorithme aussi performant que celui de [1] et travaillant sur une structure canonique.

Bien que notre implémentation n'ait pas permis d'atteindre ce but, nous nous sommes demandés si une autre optique d'implémentation comme celle expliquée dans la dernière partie du chapitre 3 n'aurait pas été plus avantageuse. Elle aurait en tout cas permis un caching des transitions le plus performant possible, moyennant une mémoire suffisante.

Cependant, c'est seulement après l'analyse des résultats obtenus et des problèmes rencontrés lors des diverses implémentations qu'il nous a été possible d'envisager cette autre solution. Notre recherche n'a donc pas été inutile car, outre le résultat d'un algorithme travaillant sur une structure canonique, nous avons également été en mesure de déterminer un nouveau type

d'implémentation qui permettrait non seulement de travailler sur une structure canonique mais également d'atteindre des temps linéaires.

Nous regrettons que le temps nous ait manqué pour réaliser nous-mêmes cette implémentation finale mais les modifications qu'elle impliquait auraient entraîné la réécriture d'une partie trop importante du code.

En ce qui concerne les apports personnels de ce mémoire, il m'a permis de réaliser une recherche de bout en bout (et d'en présenter les résultats aux 7èmes rencontres sur le parallélisme (RenPar 7) [7]). Il m'a aussi fait connaître les difficultés de telles recherches dont le résultat n'est pas toujours celui espéré. J'ai ainsi pu apprécier à leur juste valeur les résultats que ce mémoire a permis d'obtenir, bien que ces derniers ne soient pas ceux qui étaient recherchés a priori.

Finalement, j'ai été enchanté d'effectuer ce travail de fin d'étude et j'espère avoir réussi à communiquer au lecteur mon enthousiasme pour ces recherches.

Bibliographie

[1] *"An Efficient Algorithm to Compute the Synchronized Product"*
D. Zampuniéris & B. Le Charlier,
Modeling, analysis, and simulation of computer and telecommunication systems (MASCOT '95), Durham (North Carolina), January 1995.

[2] *"Efficient Handling of Large Sets of Tuples With Sharing Trees"*
D. Zampuniéris & B. Le Charlier,
Data Compression Conference (DCC '95), Snowbird (Utah), March 1995.

[3] *"Analyse Statique des Communications Entre Programmes Parallèles"*
D. Zampuniéris & B. Le Charlier,
5èmes rencontres sur le parallélisme (RenPar5), Brest (France), Mai 1993.

[4] *"Graph-based algorithms for boolean function manipulation"*
R. Bryant,
IEEE Transactions on Computers, C-35, 8:677-691, 1986.

[5] *"Generating BDD's for symbolic model checking in CCS"*
R. Enders, T. Filkorn & D. Taubner,
Distributed Computing, 6:155-164, 1993.

[6] *"Toupie : a constraint language for model checking"*
A. Rauzy,
Constraint programming : basics and trends, LNCS no. 910, Springer-Verlag, March 1995.

[7] *"Un nouvel algorithme de calcul de l'ensemble des états accessibles d'un système d'automates communicants"*
J.C. Castiaux, B. Le Charlier & D. Zampuniéris,
7ème rencontres sur le parallélisme (RenPar7), Mons (Belgique), Mai 1995.

[8] *"Communication and Concurrency"*
R. Milner
Prentice Hall, 1989.

[9] *"Construction and analysis of transition systems with MEC"*
A. Arnold, D. Bégay & P. Crubillé
A paraître, 1994

[10] *"Toupie User's Manual"* (0.25)
1994

Annexe 1
Algorithme de base

A. Code de la procédure Propagate

```
Procedure Propagate(var slistptr:PIntList;father:PItem);

 var
   s,res : PItem;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;

 begin
   while slistptr <> nil do
     begin
      { Prendre l'état de début de liste et tronquer la liste }
      num := Head(slistptr);
      oldlist := slistptr;
      slistptr := Tail(slistptr);
      dispose(oldlist);
      { Recherche du noeud correspondant à l'état num }
      s := HasSon(father,num);
      { Remise à false du champ ToBeDev }
      s^.ToBeDev := false;
      with automate[niv]^.Etat[num]^ do begin
        { Traitement des transitions asynchrones }
        for j:=1 to nbSelfTransPoss do
          begin
            { Recherche de la destination de la jème transition }
            next := SelfTransPoss[j]^.destination;
            { Correspondant de l'opération E := E U {s'} X E'
              dans l'algorithme théorique }
            nextpitem := HasSon(father,next);
            { Traitement du cas où il n'existe pas encore de fils
              représentant l'état next ~ s' dans l'algorithme théorique}
            if nextpitem = nil then
              begin
                layerptr := GetLayer(PSync2,niv);
                nextpitem := AddItem(layerptr,next);
                nextpitem^.First := false;
                AddSon(father,nextpitem);
              end;
            { Test d'inclusion des descendants de s
              dans l'ensemble des descendant de nextpitem}
            if not(Included(s, nextpitem)) then
              begin
                { E := E U {s'} x E' }
                AddDescenders(s,nextpitem);
                { Ajout du nouvel état ou de l'état modifié à la liste }
                if not nextpitem^.ToBeDev then
                  begin
                    nextpitem^.ToBeDev := TRUE;
```

```
            AddListItem(slistptr,next);
          end;
      end;
  end {for};

{ Traitement des transitions synchrones }
for j:=1 to nbSyncTransPoss do
  begin
    { Recherche du niveau de synchronisation
      de la jème transition }
    syncniv := SyncTransPoss[j]^.synchronisateur;
    { Recherche de la destination de la
      1ère transition élémentaire }
    next := SyncTransPoss[j]^.destination;
    if (syncniv > niv) then
      begin
        { Création de l'élément père pour l'appel de Transmit }
        layerptr := GetLayer(PSync2,niv);
        res := AddItem(layerptr,next);
        res^.First := false;
        oldsource := sourceniv;
        sourceniv := niv;
        { Les adresses des noeuds ne sont pas stables dans
          la boucle, il faut donc recalculer l'adresse de s
          pour être sûr qu'elle correspond bien au noeud voulu}
        s := HasSon(father,num);
        { Appel de Transmit }
        transmit(syncniv,SyncTransPoss[j],s,res);
        sourceniv := oldsource;
        { Si au moins une transition synchrone
          a pu être effectuée, càd l'arbre créé n'est pas vide }
        if res<>nil then
          begin
            { Ajout du résultat à l'arbre partagé }
            nextpitem := HasSon(father,next);
            if nextpitem = nil then
              begin
                AddSon(father,res);
                res^.ToBeDev := true;
                AddListItem(slistptr,res^.Info);
              end
            else
              begin
                { Test d'inclusion du résultat dans le sous-arbre
                  de racine le noeud fils existant nextpitem }
                if not (Included(res,nextpitem)) then
                  begin
                    { Test de l'inclusion inverse pour optimisation
                      de la mise à jour }
                    if not (Included(nextpitem,res)) then
                      AddDescenders(nextpitem,res);
                    if nextpitem^.ToBeDev = true then
                      begin
                        nextpitem^.ToBeDev := false;
                        RemoveListItem(slistptr,nextpitem^.info);
                      end;
                    { Si nécessaire, retirer l'état représenté
                      par nextpitem de la liste car nextpitem
                      va être remplacé par res }
                    if nextpitem^.ToBeDev = true then
                      begin
                        nextpitem^.ToBeDev := false;
```

```
                                RemoveListItem(slistptr,nextpitem^.info);
                              end;
                            { Remplacement du fils nextpitem par res }
                            ReplaceSon(father,nextpitem,res);
                            { Mise à jour des champs First des éléments
                              concerné (cfr. traitement de l'inclusion }
                            if nextpitem^.First then
                              res^.First := true;
                            { Si le nbre de pères de nextpitem est nul
                              on peut le supprimer de la couche }
                            if (nextpitem^.NbFathers = 0) then
                              begin
                                layerptr := GetLayer(PSync2,niv);
                                DeleteItem(layerptr,nextpitem);
                              end;
                            { Ajout du nouveau noeud à la liste }
                            res^.ToBeDev := true;
                            AddListItem(slistptr,res^.Info);
                          end;
                      end;
                  end {if res <> nil};
              end {if syncniv > niv};
          end {for};
      end {with};
    end {while};
end;
```

B. Code de la procédure Transmit

```
procedure transmit(syncniv:integer;
                   sync:PTransition;source:PItem;var res:PItem);

 var
   son : PSon;
   slistptr : PIntList;
   j,next : integer;
   resprim : PItem;

 begin
   { Initialisation de la liste slistptr }
   slistptr := nil;
   { Si on est arrivé au niveau synchrone ...}
   if (niv+1) = syncniv then
     begin
       { Analyse de tous les fils de source }
       son := source^.FirstSon;
       while (son <> nil) do
         begin
           with automate[syncniv]^.Etat[son^.Son^.Info]^ do
             begin
               for j:=1 to nbSyncTransPoss do
                 begin
                   { Si la transition cherchée est trouvée ...}
                   if (SyncTransPoss[j]^.nom = sync^.nom) and
                      (SyncTransPoss[j]^.initiateur =
                                        (not (sync^.initiateur))) and
                      (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                     begin
                       { Recherche de la destination de la transition }
                       next := SyncTransPoss[j]^.destination;
                       { Recherche du noeud éventuel correspondant }
                       layerptr := GetLayer(PSync2,syncniv);
                       nextpitem := HasSon (res,next);
                       { Si le noeud n'existe pas, on le crée }
                       if nextpitem = nil then
                         begin
                           nextpitem := AddItem(layerptr,next);
                           AddSon(res,nextpitem);
                         end ;
                       nextpitem^.First := true;
                       { Test d'inclusion des descendant de son
                         dans les descendants de nextpitem }
                       if not(Included(son^.Son, nextpitem)) then
                         begin
                           oldniv := niv;
                           niv := syncniv;
                           { Ajout des descendants de son à nextpitem }
                           AddDescenders(son^.Son,nextpitem);
                           niv := oldniv;
                           { Insertion de l'état représenté par nextpitem
                             dans slistptr, s'il ne s'y trouve pas encore }
                           if not nextpitem^.ToBeDev then
                             begin
                               nextpitem^.ToBeDev := TRUE;
                               AddListItem(slistptr,next);
                             end;
                         end;
                     end;
```

```
                end;
            end;
          son := son^.Next;
        end;
    end
  { Si on n'a pas encore atteint le niveau synchrone ...}
  else
    begin
      { Boucle sur l'ensemble des fils de source }
      son := source^.FirstSon;
      while son <> nil do
        begin
          layerptr := GetLayer(PSync2,niv+1);
          resprim := AddItem(layerptr,son^.Son^.Info);
          resprim^.First := true;
          niv := niv+1;
          { Appel de transmit récursivement }
          transmit(syncniv,sync,son^.Son,resprim);
          niv := niv-1;
          { Si le résultat de transmit n'est pas vide, ajouter
            le noeud resprim à slistptr }
          if resprim <> nil then
            begin
              AddSon(res,resprim);
              AddListItem(slistptr,resprim^.Info);
              resprim^.ToBeDev := TRUE;
            end;
          son := son^.Next;
        end;
    end;
  { Si res possède un fils, on appelle Propagate pour développer le
    niveau auquel se trouvent les fils de res }
  if (res^.FirstSon <> nil) then
    begin
      niv := niv+1;
      Propagate(slistptr,res);
      niv := niv-1;
    end
  { Si res ne possède pas de fils, aucune transition n'a pu
    être effectuée.  On signale cela en renvoyant res = nil }
  else
    begin
      layerptr := GetLayer(PSync2,niv);
      DeleteItem(layerptr,res);
      res := nil;
    end;
end;
```

C. Code des procédures ConstSync et MySynchro

```
procedure MySynchro;

var
  i     : integer;

  procedure ConstSync(i:integer; result:PItem);

  var
    slistptr : PIntList;
    item, partial : PItem;


  begin
    if i = nbautomates+1
    then begin
           { Création de la dernière couche de l'AP }
           layerptr := AddLayer(PSync2);
           { Insertion de l'élément fictif }
           item := AddItem(layerptr,-32767);
           item^.marked := false;
           item^.First := true;
           item^.ToBeDev := false;
           { Ajout de l'élément fictif comme fils de result }
           AddSon(result, item);
         end
    else begin
           { Création d'une couche supplémentaire }
           layerptr := AddLayer(PSync2);
           { Insertion du noeud correspondant à l'état initial }
           item := AddItem(layerptr,1);
           item^.marked := false;
           item^.First := true;
           item^.ToBeDev := false;
           { Ajout de ce noeud comme fils de result }
           AddSon(result, item);
           { Appel de ConstSync pour le niveau inférieur }
           ConstSync(i+1, item);
           { Insertion de item dans l'ensemble des noeuds à développer }
           item^.ToBeDev := true;
           new(slistptr);
           slistptr^.nitem := item^.Info;
           slistptr^.Next := nil;
           niv := i;
           { Réduction de l'ASP créé en AP }
           STReduce(psync2);
           { Appel de Propagate pour le développement du niveau courant }
           Propagate(slistptr,result);
         end;
  end;

begin
  i := 1;
  STInit(PSync2);
  ConstSync(i,PSync2.root);
end;
```

Annexe 2
Les optimisations

A. Cache des transitions synchrones

1. Gestion de la table de hachage

La gestion de la table de hachage MyOpTable pour le caching des transitions synchrones est effectuée par les procédures et fonctions détaillées ci-dessous :

```
{ Renvoie True si l'enregistrement de l'operation Trans sur Item est
  present dans la table de caching; False sinon }

function MyRecordedOper(Item: PItem; Trans: Ptransition;
                        NumESOK : boolean): PMyOpRecord;

var
  h: integer;
  stop: boolean;
  oprec: PMyOpRecord;
begin
  { Recherche de l'entrée correspondant à Item dans la table }
  h:= (Item^.Ident mod MaxOpTable)+1;
  stop:= false;
  MyRecordedOper:= nil;
  oprec:= MyOpTable[h];
  { Boucle sur toutes les cellules de la liste contenant les cellules
    de l'entrée h dans la table jusqu'à ce que fin de liste ou trouvé }
  while not stop do
    if oprec = nil
    then stop:= true
    else if (oprec^.IdSource = Item^.Ident)
        and (oprec^.IdTr = Trans)
        and ((oprec^.NumES = Item^.NbElements) or (not NumESOK))
        then begin
              MyRecordedOper:= oprec;
              stop:= true
             end
        else oprec:= oprec^.Next
end;
```

```
{ enregistre l'opération Trans effectuée sur Item dans
  la table de caching }

procedure MyRecordOper(Item: PItem; Trans: Ptransition;
                       DestItem: PItem);

var
  h: integer;
  oprec,tempo: PMyOpRecord;

begin
  { Recherche d'une opération semblable à celle que l'on veut enregistrer }
  oprec := MyRecordedOper(Item,Trans,false);
  { Si le résultat de la recherche est nul,
    alors on crée une nouvelle cellule et on l'insère dans la table }
  if oprec = nil then
    begin
      new(oprec);
      h:= (Item^.Ident mod MaxOpTable)+1;
      with oprec^ do begin
       IdSource:= item^.Ident;
       NumES:= item^.NbElements;
       IdTr:= Trans;
       Destptr := DestItem;
       if DestItem <> nil then
         NumED := DestItem^.NbElements
       else
         NumED := 0;
       Next:= MyOpTable[h];
      end;
      MyOpTable[h]:=oprec;
    end
  { Si le résultat de la recherche est bon,
    on remplace les informations de la cellule trouvée par les nouvelles
    informations }
  else
    begin
      with oprec^ do begin
       IdSource:= item^.Ident;
       NumES:= item^.NbElements;
       IdTr:= Trans;
       Destptr := DestItem;
       if DestItem <> nil then
         NumED := DestItem^.NbElements
       else
         NumED := 0;
      end;
    end;
end;
```

```
{ Cette fonction renvoie le résultat de l'opération Trans sur Item
  si celui-ci a été enregistré dans la table et si les informations
  qu'elle contient sont encore valides }

function GetOpResult(Item:PItem; Trans: Ptransition) : POpres;

var
  oprec : PMyOpRecord;
  dest : PItem;
  opres : POpRes;

begin
  new(opres);
  { Recherche dans la table d'une opération Trans sur Item encore valide }
  oprec := MyRecordedOper(Item,Trans,true);
  { Si une opération est trouvée,
    Vérification des paramètres de manière à assurer que l'information
    qui s'y trouve est encore valide }
  if oprec<>nil then
    begin
      dest := oprec^.destptr;
      if dest = nil then
       begin
         opres^.opOK := true;
         opres^.item := nil;
       end
      else
       { Si les résultats sont obsolètes, résultat non valide }
       if (dest^.NbElements <> oprec^.NumED) or (dest^.Ident = -1) then
         begin
           opres^.opOK := false;
           opres^.item := nil;
         end
       else
         begin
           opres^.opOK := true;
           opres^.item := dest;
         end;
    end
  { Si rien n'a été trouvé, renvoyer résultat non valide }
  else
    begin
      opres^.opOK := false;
      opres^.item := nil;
    end;
  GetOpResult := opres;
end;
```

2. Les procédures pour le comptage des éléments de sous-APs.

```
{ Renvoie le nombre d'elements dans l'arbre partage ST }
function STNbElements(var ST: SharingTree): comp;

var
  stop: boolean;
  layer: PLayer;
  item: PItem;
  s: PSon;
  trouve:boolean;
  tempo:comp;

begin
  trouve := false;
  layer:= ST.LastLayer;
  stop:= false;
  while not stop do
  begin
    if layer = nil
    then begin
          item:= ST.Root;
          stop:= true
        end
    else item:= layer^.FirstItem;
    while item <> nil do
    begin
      with item^ do
       if Info = EndOfList
       then NbElements:= 1
       else begin
              tempo := NbElements;
              NbElements:= 0;
              s:= FirstSon;
              while s <> nil do
              begin
               NbElements:= NbElements + s^.Son^.NbElements;
               s:= s^.Next
              end;
              if tempo <> NbElements then
               trouve := true;
            end;
      item:= item^.Next
    end;
    if layer <> nil
    then layer:= layer^.Previous
  end;
  STNbElements:= ST.Root^.NbElements;
  if trouve = true then
    begin
     { writeln('c''est la que ca foire');}
      {c := readkey;}
    end;
end;
```

```
procedure ItemNbelem(item:PItem);

{Cette procedure va calculer le nombre d'elements du sous-arbre ayant pour
 racine "item" en additionnant le nombre d'element des sous-arbres ayant
 pour racine les fils de "item"}

var
  son : PSon;
  nb : comp;

begin
  son := item^.FirstSon;
  nb := 0;
  { Boucle de somme de l'ensemble des elements
    des sous-APs de racine les fils de item }
  while son<>nil do
    begin
      nb := nb+son^.Son^.NbElements;
      son := son^.Next;
    end;
  item^.NbElements := nb;
end;


procedure LayerNbelem(layer:PLayer);

{Cette procedure va calculer le nombre d'elements dans les sous-arbres
 ayant pour racines les elements de la couche "layer")}

var
  item : PItem;

begin
  item := layer^.FirstItem;
  { Appel de ItemNbElem pour tous les noeuds de la couche }
  while item<>nil do
    begin
      ItemNbElem(item);
      item := item^.Next;
    end;
end;
```

3. Code des procédures Propagate, Transmit et ConstSync modifiés

```
Procedure Propagate(var slistptr:PIntList;father:PItem);

 var
   s,res : PItem;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;


 begin
   { Tant que la liste slist ptr n'est pas vide ... }
   while slistptr <> nil do
     begin
      { Extraire le premier élément de la liste
        et trouver le noeud correspondant }
      num := Head(slistptr);
      s := HasSon(father,num);
      s^.ToBeDev := false;
      oldlist := slistptr;
      slistptr := Tail(slistptr);
      dispose(oldlist);

      with automate[niv]^.Etat[num]^ do begin
        { Pour chaque transition synchrone partant du noeud courant }
        for j:=1 to nbSyncTransPoss do
          begin
            { La procedure STReduce appliquee dans cette boucle peut
              entrainer des changement d'adresse des noeuds.  D'ou le
              recalcul de la position du noeud de valeur num }
            s := HasSon(father,num);
            syncniv := SyncTransPoss[j]^.synchronisateur;
            next := SyncTransPoss[j]^.destination;
            { Si la transition est synchrone avec une transition de niveau
              inferieur }
            if (syncniv > niv) then
              begin
                layerptr := GetLayer(PSync2,niv);
                res := AddItem(layerptr,next);
                res^.First := false;
                oldsource := sourceniv;
                sourceniv := niv;
                { Appel de Transmit }
                Transmit(syncniv,SyncTransPoss[j],s,res);
                sourceniv := oldsource;
                { Si le resultat de l'appel de transmit n'est pas nul ...}
                if res<>nil then
                  begin
                    itemNbElem(res);
                    nextpitem := HasSon(father,next);
                    if nextpitem = nil then
                      begin
                        AddSon(father,res);
                        ItemNbElem(res);
                        res^.ToBeDev := true;
                        AddListItem(slistptr,res^.Info);
                      end
                    { Si father possede deja un fils de valeur next,
                      et si les descendants de res ne sont pas tous
```

```
            descendants de ce fils,
            alors remplacer ce fils par res et ajouter a res
            l'ensemble des descendant de ce fils }
          else
            begin
              if not (Included(res,nextpitem)) then
                begin
                  if not (Included(nextpitem,res)) then
                    AddDescenders(nextpitem,res);
                  ItemNbElem(res);
                  if nextpitem^.ToBeDev = true then
                    begin
                      nextpitem^.ToBeDev := false;
                      RemoveListItem(slistptr,nextpitem^.info);
                    end;
                  ReplaceSon(father,nextpitem,res);
                  if nextpitem^.First then
                    res^.First := true;
                  if (nextpitem^.NbFathers = 0) then
                    begin
                      layerptr := GetLayer(PSync2,niv);
                      DeleteItem(layerptr,nextpitem);
                    end;
                  res^.ToBeDev := true;
                  AddListItem(slistptr,res^.Info);
                end;
            end;
          if c='a' then
            begin
              STReduce(psync2);
              c:='b';
            end
          else
            if c='b' then
              c:='c'
            else
              if c='c' then
                c:='d'
              else
                c:='a';
        end;
    end;
  end;

{ Boucle sur l'ensemble des transitions asynchrones
  partant du noeud courant }

for j:=1 to nbSelfTransPoss do
  begin
    { Recherche de la destination de la transition }
    next := SelfTransPoss[j]^.destination;
    nextpitem := HasSon(father,next);
    { Si father ne possede pas encore de fils de valeur next }
    if nextpitem = nil then
      begin
        layerptr := GetLayer(PSync2,niv);
        nextpitem := AddItem(layerptr,next);
        nextpitem^.First := false;
        AddSon(father,nextpitem);
      end;
    { Si l'ensemble des descendants de s n'est pas inclu dans
      l'ensemble des descendants de nextpitem ...}
    if not(Included(s, nextpitem)) then
      begin
```

```
                { Ajout des descendants de s a nextpitem }
                AddDescenders(s,nextpitem);
                ItemNbElem(nextpitem);
                if not nextpitem^.ToBeDev then
                  begin
                    nextpitem^.ToBeDev := TRUE;
                    AddListItem(slistptr,next);
                  end;
              end;
          end;
      end;
    end;
end;


procedure Transmit(syncniv:integer;
                   sync:PTransition;source:PItem;var res:PItem);

var
  son : PSon;
  slistptr : PIntList;
  j,next : integer;
  resprim : PItem;

begin
  { Initialisation de la variable slistptr }
  slistptr := nil;
  { Si on se trouve au niveau synchrone, ... }
  if (niv+1) = syncniv then
    begin
      { Boucle sur l'ensemble des fils de source }
      son := source^.FirstSon;
      while (son <> nil) do
        begin
          with automate[syncniv]^.Etat[son^.Son^.Info]^ do
            begin
              { Boucle sur l'ensemble des transitions synchrones
                d'origine le fils courant }
              for j:=1 to nbSyncTransPoss do
                begin
                  { Si c'est la transition recherchée, ... }
                  if (SyncTransPoss[j]^.nom = sync^.nom) and
                     (SyncTransPoss[j]^.initiateur =
                                        (not (sync^.initiateur))) and
                     (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                    begin
                      { Libérer eventuellement le pointeur opresult }
                      if opresult <> nil then
                        dispose (opresult);
                      { RecTrans determine s'il faut utiliser le caching }
                      if not RecTrans then
                        begin
                          { Si pas de caching,
                            Alors mettre opresult^.opOk a false }
                          new(opresult);
                          opresult^.item := nil;
                          opresult^.opOK := false;
                        end
                      else
                          { Recherche de l'enregistrement d'une operation
                            sync sur son^.Son }
                          opresult := GetOpResult(son^.Son,sync);
```

```
nextpitem := opresult^.item;
next := SyncTransPoss[j]^.destination;
{ Si aucune operation enregistree n'a ete trouvee,
  alors ... }
if not opresult^.opOK then
  begin
    layerptr := GetLayer(PSync2,syncniv);
    nextpitem := HasSon (res,next);
    { Si res ne possede pas de fils de valeur res }
    if nextpitem = nil then
      begin
        nextpitem := AddItem(layerptr,next);
        AddSon(res,nextpitem);
      end;
    nextpitem^.First := true;
    { Si les descendants de son^.Son ne sont pas
      inclus dans les descendants de nextpitem }
    if not(Included(son^.Son, nextpitem)) then
      begin
        { Ajout des descendants du fils courant a
          nextpitem }
        oldniv := niv;
        niv := syncniv;
        AddDescenders(son^.Son,nextpitem);
        niv := oldniv;
        { Ajout de nextpitem dans la liste
          des noeuds a developper }
        if not nextpitem^.ToBeDev then
          begin
            nextpitem^.ToBeDev := TRUE;
            AddListItem(slistptr,next);
          end;
        ItemNbElem(nextpitem);
      end;
    { Enregistrement de l'operation effectuee }
    if RecTrans then
       MyRecordOper(son^.Son,sync,nextpitem);
  end
else
  { S'il y a caching de la transition... }
  if nextpitem <> nil then
    begin
      { Ajouter le resultat à l'ensemble des
        descendants de res }
      copitem := HasSon(res,nextpitem^.Info);
      if copitem = nil then
        begin
          AddSon(res,nextpitem);
          if not nextpitem^.ToBeDev then
            begin
              nextpitem^.ToBeDev := TRUE;
              AddListItem(slistptr,next);
            end;
        end
      else
        begin
          if not Included(nextpitem,copitem) then
            begin
              { Ajout des descendants de nextpitem
                a copitem }
              oldniv := niv;
              niv := syncniv;
              AddDescenders(nextpitem,copitem);
```

```
                                    niv := oldniv;
                                    { Ajout de copitem dans la liste
                                      des noeuds a developper }
                                    if not copitem^.ToBeDev then
                                      begin
                                        copitem^.ToBeDev := TRUE;
                                        AddListItem(slistptr,next);
                                      end;
                                    ItemNbElem(nextpitem);
                                  end;
                              end;
                          end;

                    end;
                end;
            end;
          son := son^.Next;
        end;
    end
{ Si on se trouve dans un niveau intermediaire ...}
else
  begin
    { Boucle sur l'ensemble des fils de source }
    son := source^.FirstSon;
    while son <> nil do
      begin
        { Creation d'un nouvel element avant l'appel de Transmit }
        layerptr := GetLayer(PSync2,niv+1);
        resprim := AddItem(layerptr,son^.Son^.Info);
        resprim^.First := true;
        if opresult <> nil then
          dispose(opresult);
        { Si on demande de ne pas se servir du caching ...}
        if not RecTrans then
          begin
            { Mettre le champ opOk de opresult a false }
            new(opresult);
            opresult^.opOK := false;
            opresult^.item := nil;
          end
        else
            { Recherche de l'enregistrement d'une operation
              sync sur son^.Son }
            opresult := GetOpResult(son^.Son,sync);
        nextpitem := opresult^.item;
        { Si le resultat de la recherche est nul ... }
        if not opresult^.opOK then
          begin
            { Appel de Transmit recursivement }
            niv := niv+1;
            transmit(syncniv,sync,son^.Son,resprim);
            niv := niv-1;
            { Enregistrement de l'operation }
            if RecTrans then
                MyRecordOper(son^.Son,sync,resprim);
            { Si le resultat de l'appel de Transmit n'est pas nul ...}
            if resprim <> nil then
              begin
                ItemNbElem(resprim);
                { Ajout de resprim comme fils de res }
                AddSon(res,resprim);
                AddListItem(slistptr,resprim^.Info);
                resprim^.ToBeDev := TRUE;
```

```
                end;
              end
            { Si le resultat de la recherche n'est pas nul ... }
            else
              if nextpitem <> nil then
                  { Ajouter nextpitem (resultat de la recherche)
                    comme fils de res }
                  AddSon(res,nextpitem);
          end;
      end;
    { Si au moins un fils a ete cree <=> slistptr n'est pas vide }
    if (res^.FirstSon <> nil) then
      begin
        { Appel de Propagate pour le developpement des fils de res }
        niv := niv+1;
        Propagate(slistptr,res);
        niv := niv-1;
      end
    { Si aucune operation n'a pu etre faite sur les fils de res,
      Alors il faut renvoyer res = nil }
    else
      begin
        layerptr := GetLayer(PSync2,niv);
        DeleteItem(layerptr,res);
        res := nil;
      end;
  end;

procedure MySynchro;

var
  i     : integer;

  procedure ConstSync(i:integer; result:PItem);

  var
    slistptr : PIntList;
    item, partial : PItem;


  begin
    { Si i = nbautomates+1,
      Alors creation d'un etat fictif extremite de l'arbre }
    if i = nbautomates+1
    then begin
           layerptr := AddLayer(PSync2);
           item := AddItem(layerptr,-32767);
           item^.marked := false;
           item^.First := true;
           item^.ToBeDev := false;
           item^.NbElements := 1;
           { Ajout du noeud fictif comme fils de result }
           AddSon(result, item);
           ItemNbElem(result);
         end
    { Sinon, creation de l'etat initial de l'automate i
      et appel de Propagate pour le developpement }
    else begin
           { Creation d'un nouveau noeud
             representant l'etat initial de l'automate i }
           layerptr := AddLayer(PSync2);
```

```
            item := AddItem(layerptr,1);
            item^.marked := false;
            item^.First := true;
            item^.ToBeDev := false;
            item^.NbElements := 1;
            AddSon(result, item);
            ItemNbElem(result);
            { Connection du resultat
              a l'etat initial de l'automate superieur }
            ConstSync(i+1, item);
            if item^.FirstSon = nil then
            item^.ToBeDev := true;
            new(slistptr);
            slistptr^.nitem := item^.Info;
            slistptr^.Next := nil;
            niv := i;
            STReduce(psync2);
            ItemNbElem(item);
            { Appel de Propagate pour le developpement du niveau i }
            Propagate(slistptr,result);
          end;
  end;

begin
  i := 1;
  STInit(PSync2);
  ConstSync(i,PSync2.root);
end;
```

B. Réduction dynamique sans backtracking

1. Procédures et fonctions de gestion de la table de hachage TreeTable

```
{ Recherche dans la table de hachage des arbres un sous-AP ayant
  comme racine un noeud de valeur info et comme fis l'ensemble des
  noeud compris dans itemarray }

Function SearchTable(info,nbr:integer;itemarray:TItemArray) : PItem;

var
h,i,mini : integer;
rectree : PTreeRecord;
sonptr : PListItem;
trouve,stop,compok : boolean;

begin
  { Calcul de la position de l'eventuel enregistrement dans la table }
  if nbr >= 3 then
    mini:=3
  else
    mini:=nbr;
  h := ((itemarray[(1 mod mini)+1]^.Ident) div 3 +
       (itemarray[(2 mod mini)+1]^.Ident) div 3 +
       (itemarray[(3 mod mini)+1]^.Ident) div 3) mod MaxTreeTable +1;
  stop:= false;
  trouve := false;
  compok := true;
  rectree := TreeTable[h];
  { Parcours de la liste de l'entree h de la table pour la recherche
    du sous-AP }
  while not stop do
    if rectree = nil
    then stop:= true
    else
      { Test : La valeur de la racine doit etre Info }
      if (rectree^.InfSource = Info) then
       begin
         sonptr := rectree^.ListSonItem;
         i := 1;
         compok := true;
         { Comparaison des fils respectifs
           (classe par ordre croissant de info) }
         while (i<=nbr) and (sonptr <> nil) do
           begin
             if itemarray[i] = sonptr^.Item then
              begin
                i := i+1;
                sonptr := sonptr^.Next;
              end
             else
              begin
                sonptr := nil;
                compok := false;
              end;
           end;
         { Si la comparaison donne un resultat positif et si tous les
           fils ont ete pris en consideration, alors on a trouve,
           sinon, on passe a la cellule suivante }
         if compok then
           if (i <> nbr+1) or (sonptr <> nil) then
```

```
            rectree := rectree^.Next
          else
            begin
             trouve := true;
             stop := true;
            end
        else
          rectree := rectree^.Next;
      end
     else
      rectree := rectree^.Next;

  { Si le resultat de la recherche est positif,
    alors renvoyer la racine de l'arbre trouve }

  if trouve then
    SearchTable := rectree^.sourceptr
  else
    SearchTable := nil;

end;


{ Cette procedure supprime de la table la cellule correspondant a
  l'enregistrement du sous-AP de racine x }

procedure RemoveFromTable(x : PItem);

var
h,nbr : integer;
prec,rectree : PTreeRecord;
stop : boolean;
preccell,cellptr : PListItem;
sonptr : PSon;

begin
  { Recherche de l'entree de la table TreeTable correspondant
    au sous-AP de racine x }
  sonptr := x^.FirstSon;
  nbr := 0;
  while (nbr<3) and (sonptr<>nil) do
    begin
      nbr := nbr+1;
      itar[nbr] := sonptr^.Son;
      sonptr := sonptr^.Next;
    end;
  h := ((itar[(1 mod nbr)+1]^.Ident) div 3 +
       (itar[(2 mod nbr)+1]^.Ident) div 3 +
       (itar[(3 mod nbr)+1]^.Ident) div 3) mod MaxTreeTable +1;
  stop:= false;
  rectree := TreeTable[h];
  prec := nil;
  { Recherche de l'enregistrement correspondant a x dans la liste }
  while not stop do
    if rectree = nil
    then stop:= true
    else
      { Si l'enregistrement est trouve, alors on supprime la cellule }
      if (rectree^.Sourceptr = x) then
       begin
         if prec = nil then
           begin
```

```
            TreeTable[h] := rectree^.next;
            stop := true
          end
        else
          begin
            prec^.Next := rectree^.Next;
            stop := true
          end;
        Cellptr := rectree^.ListSonItem;
        while cellptr <> nil do
          begin
            preccell := cellptr;
            cellptr := cellptr^.Next;
            dispose(preccell);
          end;
        dispose(rectree);
      end
      { Si l'enregistrement n'est pas trouve,
       Alors on passe a la cellule suivante }
      else
       begin
         prec := rectree;
         rectree := rectree^.Next;
       end;
end;


{ Cette procedure insere une cellule correspondant aux informations
  concernant le sous-AP de racine x }

Procedure InsertInTable(x : PItem);

var h,nbr : integer;
    id : integer;
    trouve : boolean;
    oldrectree,rectree : PTreeRecord;
    ptrson : PSon;
    ptritem,prec : PListItem;

begin

  { Recherche de l'entree correspondante dans la table TreeTable }
  ptrson := x^.FirstSon;
  nbr := 0;
  while (nbr<3) and (ptrson<>nil) do
    begin
      nbr := nbr+1;
      itar[nbr] := ptrson^.Son;
      ptrson := ptrson^.Next;
    end;
  h := ((itar[(1 mod nbr)+1]^.Ident) div 3 +
       (itar[(2 mod nbr)+1]^.Ident) div 3 +
       (itar[(3 mod nbr)+1]^.Ident) div 3) mod MaxTreeTable +1;
  trouve := false;
  rectree := TreeTable[h];
  if trouve = false then
    begin
      rectree := TreeTable[h];;
      new(TreeTable[h]);
      TreeTable[h]^.InfSource := x^.Info;
      TreeTable[h]^.Sourceptr := x;
      TreeTable[h]^.Next := rectree;
```

```
      rectree := TreeTable[h];
      ptrson := x^.FirstSon;
      if ptrson <> nil then
       begin
         new(rectree^.ListSonItem);
         rectree^.ListSonItem^.Item := ptrson^.Son;
         ptritem := rectree^.ListSonItem;
         ptritem^.Next := nil;
         ptrson := ptrson^.Next;
       end;
      while ptrson <> nil do
       begin
         new(ptritem^.Next);
         ptritem := ptritem^.Next;
         ptritem^.Item := ptrson^.Son;
         ptritem^.Next := nil;
         ptrson := ptrson^.Next;
       end;
    end;
end;
```

2. Code de la procédure Propagate.

```
Procedure Propagate(var slistptr:PIntList;var father:PItem);

 var
   copfather : PItem;
   s,res,newfather : PItem;
   sonptr : PSon;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;
   inserted : boolean;


 begin
   { Si niv > 1, creation d'une copie de father }
   if niv > 1 then
     begin
       layerptr := GetLayer(PSync2,niv-1);
       copfather := AddItem(layerptr,father^.Info);
       sonptr := father^.FirstSon;
       while sonptr <> nil do
         begin
           AddSon(copfather,sonptr^.Son);
           sonptr := sonptr^.Next;
         end;
     end
   else
     copfather := father;

   while slistptr <> nil do
     begin
      { Choisir le premier element de la liste slistptr }
      num := Head(slistptr);
      s := HasSon(copfather,num);
      s^.ToBeDev := false;
      { Enlever le premier element de la liste slistptr }
      oldlist := slistptr;
      slistptr := Tail(slistptr);
      dispose(oldlist);
      with automate[niv]^.Etat[num]^ do begin
        { Pour chaque transition synchrone partant de l'element s...}
        for j:=1 to nbSyncTransPoss do
          begin
            s := HasSon(copfather,num);
            syncniv := SyncTransPoss[j]^.synchronisateur;
            next := SyncTransPoss[j]^.destination;
            { Si le niveau synchrone se trouve
              dans une couche plus basse de l'arbre ...}
            if (syncniv > niv) then
              begin
                layerptr := GetLayer(PSync2,niv);
                oldsource := sourceniv;
                sourceniv := niv;
                { Appel de transmit }
                transmit(syncniv,SyncTransPoss[j],s,res,next);
                sourceniv := oldsource;
                { Si la transition synchrone a pu etre effectuee ...}
                if res<>nil then
                  begin
```

```
nextpitem := HasSon(copfather,next);
{ S'il n'y avait pas encore de fils de valeur next...}
if nextpitem = nil then
  begin
    addson(copfather,res);
    res^.ToBeDev := true;
    AddListItem(slistptr,res^.Info);
  end
{ S'il existe deja un fils de valeur next...}
else
  begin
    { Si le sous-arbre res n'est pas inclu
      dans nextpitem...}
    if not (Included(res,nextpitem)) then
      begin
        { Si le sous-arbre nextpitem n'est pas inclu
          dans res...}
        if not (Included(nextpitem,res)) then
          begin
            { Calculer l'union de nextpitem
              et de res...}
            newson := myunion(nextpitem,res,true);
            replaceson(copfather,nextpitem,newson);
            { Enlever le noeud nextpitem
              de la liste s'il y est }
            if nextpitem^.ToBeDev = true then
             begin
              nextpitem^.ToBeDev := false;
              RemoveListItem(slistptr,nextpitem^.info);
             end;
            { Supprimer le noeud nextpitem
              s'il n'a plus de pere }
            if (nextpitem^.Nbfathers = 0) then
              begin
                layerptr := GetLayer(PSync2,niv);
                RemoveFromTable(nextpitem);
                DeleteItem(layerptr,nextpitem);
              end;
            end
        else
          begin
            { Remplacer nextpitem par newson }
            newson := res;
            ReplaceSon(copfather,nextpitem,newson);
            { Si nextpitem est dans la liste,
              l'enlever }
            if nextpitem^.ToBeDev = true then
             begin
              nextpitem^.ToBeDev := false;
              RemoveListItem(slistptr,nextpitem^.info);
             end;
          end;
        { Si res n'a pas de pere, alors le supprimer }
        if (res^.Nbfathers = 0) then
          begin
            layerptr := GetLayer(PSync2,niv);
            RemoveFromTable(res);
            DeleteItem(layerptr,res);
          end;
        { Ajouter newson a l'ensemble
          des noeuds a developper }
        newson^.ToBeDev := true;
        AddListItem(slistptr,newson^.Info);
```

```
                end;
            end;
        end;
    end;
  end;

{ Pour toutes les transitions "selfs" partant de s ...}
for j:=1 to nbSelfTransPoss do
  begin
    next := SelfTransPoss[j]^.destination;
    nextpitem := HasSon(copfather,next);
    { S'il n'existe pas encore de fils de valeur next ...}
    if nextpitem = nil then
      begin
        layerptr := GetLayer(PSync2,niv);
        { Recherche d'un sous-AP correspondant
          au resultat de la transition }
        nb := 0;
        son := s^.FirstSon;
        while son <> nil do
          begin
            nb := nb+1;
            itemarray[nb] := son^.Son;
            son := son^.Next;
          end;
        nextpitem := SearchTable(next,nb,itemarray);
        { Si le resultat de la recherche est nul }
        if nextpitem = nil then
          begin
            { Creer un nouvel element et lui ajouter l'ensemble
              des descendants de s.  Ajouter ce nouvel element
              comme fils de copfather }
            nextpitem := AddItem(layerptr,next);
            nextpitem^.First := false;
            AddSon(copfather,nextpitem);
            AddDescenders(s,nextpitem);
            { Inserer le nouveau sous-AP dans la table }
            InsertInTable(nextpitem);
            { Ajouter le nouveau noeud a l'ensemble
              des noeuds a developper }
            if nextpitem^.ToBeDev = false then
              begin
                nextpitem^.ToBeDev := true;
                AddListItem(slistptr,nextpitem^.Info);
              end;
          end
        { Si le resultat de la recherche n'est pas nul ... }
        else
          begin
            { Ajouter le resultat comme fils de copfather }
            AddSon(copfather,nextpitem);
            { Ajouter nextpitem a l'ensemble
              des noeuds a developper }
            if nextpitem^.ToBeDev = false then
              begin
                nextpitem^.ToBeDev := true;
                AddListItem(slistptr,nextpitem^.Info);
              end;
          end;
      end
    { Si copfather possede deja un fils de valeur next ...}
    else
      if not(Included(s, nextpitem)) then
```

```
            begin
              { Rechercher ou construire un noeud de valeur next
                ayant pour descendants l'union des descendants
                de s et de nextpitem }
              newson := myunion(s,nextpitem,true);
              { Si un sous-AP correspondant a ete trouve
                => newson <> nextpitem }
              if newson <> nextpitem then
                begin
                  { Remplacer le fils nextpitem par newson }
                  ReplaceSon(copfather,nextpitem,newson);
                  { Si nextpitem n'a plus de pere, le supprimer }
                  if nextpitem^.Nbfathers = 0 then
                    begin
                      RemoveFromTable(nextpitem);
                      if nextpitem^.ToBeDev then
                        RemoveListItem(slistptr,nextpitem^.info);
                      layerptr := GetLayer(PSync2,niv);
                      DeleteItem(layerptr,nextpitem);
                    end;
                end;
              { Ajouter newson a l'ensemble des noeuds a developper }
              if not newson^.ToBeDev then
                begin
                  newson^.ToBeDev := TRUE;
                  AddListItem(slistptr,next);
                end;
            end;
        end;
    end;

  end;

{ Traitement de fin de procedure pour assurer la reduction du graphe }


{ Recherche d'un sous-AP equivalent
  au sous-AP de racine copfather construit }
sonptr := copfather^.FirstSon;
nb := 0;
while sonptr <> nil do
  begin
    nb := nb+1;
    itemarray[nb] := sonptr^.Son;
    sonptr := sonptr^.Next;
  end;
newfather := SearchTable(copfather^.Info,nb,itemarray);
{ Si le resultat de la recherche est nul, ...}
if newfather = nil then
  begin
    { Modifier father en tenant compte des ajouts fait dans copfather }
    RemoveFromTable(father);
    AddDescenders(copfather,father);
    InsertInTable(father);
    if (father <> copfather) and (niv > 1) then
      begin
        layerptr := GetLayer(PSync2,niv-1);
        DeleteItem(layerptr,copfather);
      end;
  end
{ Si le resultat de la recherche n'est pas nul ... }
else
  begin
```

```
      { S'il y a eu copie,
        c'est-a-dire si on ne se trouve pas avec father=root }
      if niv > 1 then
        begin
          { Si le sous-AP trouve n'est pas le meme
            que le sous-AP de racine copfather }
          if newfather <> copfather then
            begin
              layerptr := GetLayer(PSync2,niv-1);
              DeleteItem(layerptr,copfather);
            end;
          father := newfather;
        end;
    end;
end;
```

3. Code de la procédure Transmit modifiée.

```
procedure Transmit(syncniv:integer;
                   sync:PTransition;source:PItem;
                   var res:PItem;resval:integer);

 var
   inserted : boolean;
   son : PSon;
   slistptr : PIntList;
   i,j,next : integer;
   resprim : PItem;
   rescree : boolean;

 begin
   slistptr := nil;
   rescree := false;
   { Si on se trouve au niveau synchrone ... }
   if (niv+1) = syncniv then
     begin
       { Boucle sur l'ensemble des fils de source }
       son := source^.FirstSon;
       while (son <> nil) do
         begin
           with automate[syncniv]^.Etat[son^.Son^.Info]^ do
             begin
               for j:=1 to nbSyncTransPoss do
                 begin
                   if (SyncTransPoss[j]^.nom = sync^.nom) and
                      (SyncTransPoss[j]^.initiateur =
                                        (not (sync^.initiateur))) and
                     (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                     begin
                       { L'option RecTrans n'a pas ete modifiee
                         sur cette version }
                       if opresult <> nil then
                         dispose (opresult);
                       if not RecTrans then
                         begin
                           new(opresult);
                           opresult^.item := nil;
                           opresult^.opOK := false;
                         end
                       else
                           { Jamais accede }
                           opresult := GetOpResult(son^.Son,sync);

                       nextpitem := opresult^.item;
                       next := SyncTransPoss[j]^.destination;
                       { Toujours not opresult^.opOK = true }
                       if not opresult^.opOK then
                         begin
                           layerptr := GetLayer(PSync2,syncniv);
                           if rescree then
                             nextpitem := HasSon (res,next)
                           else
                             nextpitem := nil;
                           { Si res n'a pas encore de fils de valeur next
                             ou si res n'a pas été créé ...}
                           if nextpitem = nil then
                             begin
                               { Recherche d'un arbre équivalent
```

```
  à celui à construire }
nb := 0;
ltson := son^.Son^.FirstSon;
while ltson <> nil do
  begin
    nb := nb+1;
    itemarray[nb] := ltson^.Son;
    ltson := ltson^.Next;
  end;
nextpitem := SearchTable(next,nb,itemarray);
{ Si aucun arbre n'a été trouvé...}
if nextpitem = nil then
  begin
    { Construction d'un nouveau noeud,
      fils de res et ajout des descendants
      de nextpitem }
    nextpitem := AddItem(layerptr,next);
    nextpitem^.First := true;
    oldniv := niv;
    niv := syncniv;
    AddDescenders(son^.Son,nextpitem);
    niv := oldniv;
    nextpitem^.First := true;
    { Insertion du nouveau noeud
      dans la table }
    InsertInTable(nextpitem);
    if not rescree then
      begin
        layerptr := Getlayer(PSync2,niv);
        res := AddItem(layerptr,resval);
        res^.First := true;
        rescree := true;
      end;
    AddSon(res,nextpitem);
    { Ajout de nextpitem a l'ensemble des
      noeuds a developper }
    if not nextpitem^.ToBeDev then
      begin
        nextpitem^.ToBeDev := TRUE;
        AddListItem(slistptr,next);
      end;
  end
{ Si le resultat de la recherche
  n'est pas nul }
else
  begin
    { Si le noeud res n'existe pas encore }
    if not rescree then
      begin
        { Creer un noeud res }
        layerptr := GetLayer(PSync2,niv);
        res := AddItem(layerptr,resval);
        res^.First := true;
        rescree := true;
        { Ajouter nextpitem comme fils
          de res }
        AddSon(res,nextpitem);
        nextpitem^.First := true;
        { Ajouter nextpitem a l'ensemble des
          noeuds a developper }
        if not nextpitem^.ToBeDev then
          begin
            nextpitem^.ToBeDev := TRUE;
```

```
                    AddListItem(slistptr,next);
                  end;
              end
            { Si res a deja ete cree }
            else
              begin
                { Ajouter nextpitem comme fils
                  de res }
                AddSon(res,nextpitem);
                nextpitem^.First := true;
                { Ajouter nextpitem a l'ensemble
                  des noeuds a developper }
                if not nextpitem^.ToBeDev then
                  begin
                    nextpitem^.ToBeDev := TRUE;
                    AddListItem(slistptr,next);
                  end;
              end;
          end;
      end
    { Si res possede deja un fils de valeur next }
    else
      begin
        { Recherche ou construction d'un noeud
          union de son^.Son et nextpitem
          et de valeur next }
        if res^.Nbfathers = 0 then
          newson := myunion(son^.Son,nextpitem,true)
        else
          newson:=myunion(son^.Son,nextpitem,false);
        newson^.First := true;
        { Si le nouveau noeud n'est pas nextpitem }
        if newson <> nextpitem then
          begin
            { Remplacer nextpitem par newson
              dans l'ensemble des fils de res }
            ReplaceSon(res,nextpitem,newson);
            if nextpitem^.Nbfathers = 0 then
              begin
                RemoveFromTable(nextpitem);
                layerptr:= Getlayer(PSync2,syncniv);
                DeleteItem(layerptr,nextpitem);
              end;
            { Ajout de newson a l'ensemble
              des noeuds a developper }
            if not newson^.ToBeDev then
              begin
                newson^.ToBeDev := TRUE;
                AddListItem(slistptr,next);
              end;
          end;
      end;
    { Jamais RecTrans }
    if RecTrans then
       MyRecordOper(son^.Son,sync,nextpitem);
  end
{ Pas de caching }
else
  { Jamais par ici }
  if nextpitem <> nil then
    begin
      {Pas encore modifie en vue
       de seconde optimisation}
```

```
                        AddSon(res,nextpitem);
                        if not nextpitem^.ToBeDev then
                          begin
                            nextpitem^.ToBeDev := TRUE;
                            AddListItem(slistptr,next);
                          end;
                        ItemNbElem(res);
                      end;
                  end;
              end;
            son := son^.Next;
        end;
    end
{ Niveau intermediaire }
else
  begin
    { Boucle sur les fils de source }
    son := source^.FirstSon;
    while son <> nil do
      begin
        layerptr := GetLayer(PSync2,niv+1);
        if opresult <> nil then
          dispose(opresult);
        { Toujours not RecTrans }
        if not RecTrans then
          begin
            new(opresult);
            opresult^.opOK := false;
            opresult^.item := nil;
          end
        else
            opresult := GetOpResult(son^.Son,sync);
        nextpitem := opresult^.item;
        { Toujours not opresult^.opOk car caching non implemente
          sur cette version }
        if not opresult^.opOK then
          begin
            { Appel de Transmit recursivement }
            niv := niv+1;
            transmit(syncniv,sync,son^.Son,resprim,son^.Son^.Info);
            niv := niv-1;
            { RecTrans = false car pas de caching autorise }
            if RecTrans then
                MyRecordOper(son^.Son,sync,resprim);
            { Si le resultat de Transmit n'est pas nul }
            if resprim <> nil then
              begin
                { Si res est cree, ajouter resprim comme fils }
                if rescree then
                  AddSon(res,resprim)
                else
                  begin
                    { Creer res et ajouter resprim comme fils }
                    layerptr := GetLayer(PSync2,niv);
                    res := AddItem(layerptr,resval);
                    res^.First := true;
                    rescree := true;
                    AddSon(res,resprim);
                  end;
                { Ajouter resprim a l'ensemble des noeuds a developper }
                AddListItem(slistptr,resprim^.Info);
                resprim^.ToBeDev := TRUE;
```

```
            end;
          end
        { Jamais car pas de caching autorise }
        else
          if nextpitem <> nil then
            begin
              {Pas encore modifie pour la seconde optimisation}
              AddSon(res,nextpitem);
              if nextpitem^.First = false then
                begin
                  writeln('nextpitem^.first=false a la fin');
                  c := readkey;
                end;
              layerptr := GetLayer(PSync2,niv);
              LayerNbElem(layerptr);
            end;
        son := son^.Next;
      end;
  end;
 { Si res a ete cree et possede des fils
   a developper ... }
 if rescree then
   if (res^.FirstSon <> nil) then
     begin
       { Appel de Propagate }
       niv := niv+1;
       Propagate(slistptr,res);
       niv := niv-1;
       { RecTrans = false car pas de caching autorise }
       if RecTrans then
         begin
           son := source^.FirstSon;
           while son <> nil do
             begin
               MyUpdateOper(son^.Son,sync);
               son := son^.Next;
             end;
         end;
     end
   { Si res ne possede pas de fils, renvoyer res = nil }
   else
     begin
       RemoveFromTable(res);
       layerptr := GetLayer(PSync2,niv);
       DeleteItem(layerptr,res);
       res := nil;
     end
{ Si res n'a pas ete cree, Renvoyer res = nil }
else
  res := nil;
end;
```

C. *Réduction dynamique avec BackTracking et optimisation combinee.*

1. Code de la procédure Propagate modifiée.

```
Procedure Propagate(var slistptr:PIntList;var father:PItem);

 var
   s,res,newfather : PItem;
   sonptr : PSon;
   num,next : integer;
   j        : integer;
   syncniv  : integer;
   oldlist : PIntList;
   oldsource : integer;
   inserted : boolean;


 begin
   while slistptr <> nil do
     begin
      { Choisir le premier element de la liste slistptr }
      num := Head(slistptr);
      s := HasSon(father,num);
      s^.ToBeDev := false;
      { Enlever le premier element de la liste slistptr }
      oldlist := slistptr;
      slistptr := Tail(slistptr);
      dispose(oldlist);
      with automate[niv]^.Etat[num]^ do begin
      { Pour chaque transition synchrone partant de l'element s...}
      for j:=1 to nbSyncTransPoss do
        begin
          s := HasSon(father,num);
          syncniv := SyncTransPoss[j]^.synchronisateur;
          next := SyncTransPoss[j]^.destination;
          { Si le niveau synchrone se trouve
            dans une couche plus basse de l'arbre ...}
          if (syncniv > niv) then
            begin
              layerptr := GetLayer(PSync2,niv);
              oldsource := sourceniv;
              sourceniv := niv;
              { Appel de transmit }
              transmit(syncniv,SyncTransPoss[j],s,res,next);
              sourceniv := oldsource;
              { Si la transition synchrone a pu etre effectuee ...}
              if res<>nil then
                begin
                  ItemNbElem(res);
                  nextpitem := HasSon(father,next);
                  { S'il n'y avait pas encore de fils de valeur next...}
                  if nextpitem = nil then
                    begin
                      { Recherche d'un sous-AP dont la racine a la
                        valeur de father et ayant pour fils l'ensemble
                        des fils de father + res }
                      nb := 0;
                      inserted := false;
                      sonptr := father^.FirstSon;
```

```
    while sonptr <> nil do
      begin
        nb := nb+1;
        if (sonptr^.Son^.Info > res^.Info)
            and (not inserted) then
          begin
            itemarray[nb] := res;
            inserted := true;
          end
        else
          begin
            itemarray[nb] := sonptr^.Son;
            sonptr := sonptr^.Next;
          end;
      end;
    if not inserted then
      begin
        nb := nb+1;
        itemarray[nb] := res;
      end;
    newfather := SearchTable(father^.Info,nb,itemarray);
    { Si le resultat de la recherche est nul }
    if newfather = nil then
      begin
        RemoveFromTable(father);
        AddSon(father,res);
        InsertInTable(father);
        res^.ToBeDev := true;
        AddListItem(slistptr,res^.Info);
      end
    { Sinon ... }
    else
      begin
        res^.ToBeDev := true;
        AddListItem(slistptr,res^.Info);
        { Modification des niveaux superieurs
          de maniere a eviter les redondances
          dues aux remplacement de father par
          newfather }
        Backtracking(father,newfather);
        father := newfather;
      end;
  end
{ S'il existe deja un fils de valeur next...}
else
  begin
    { Si le sous-arbre res n'est pas inclu
      dans nextpitem...}
    if not (Included(res,nextpitem)) then
      begin
        { Si le sous-arbre nextpitem n'est pas
          inclu dans res...}
        if not (Included(nextpitem,res)) then
          begin
            { Calculer l'union de nextpitem et de res...}
            newson := myunion(nextpitem,res,true);
            ItemNbElem(newson);
            if newson <> res then
              begin
                { La modification de l'ensemble des fils
                  de father doit être propagee en arriere
                  de maniere a verifier que l'on
                  n'introduit pas une redondance dans le
```

```
            sharing tree }
          sonptr := father^.FirstSon;
          nb := 0;
          while sonptr <> nil do
            begin
              nb := nb+1;
              if sonptr^.Son^.Info= newson^.Info then
                itemarray[nb] := newson
              else
                itemarray[nb] := sonptr^.Son;
              sonptr := sonptr^.Next;
            end;
          if nextpitem^.ToBeDev = true then
           begin
            nextpitem^.ToBeDev := false;
            RemoveListItem(slistptr,nextpitem^.info);
           end;
          Backtracking(nextpitem,newson);
          father :=
             SearchTable(father^.Info,nb,itemarray);
        end
          { newson = res }
      else
        begin
          RemoveFromTable(father);
          ReplaceSon(father,nextpitem,newson);
          InsertInTable(father);
          if nextpitem^.ToBeDev = true then
            begin
              nextpitem^.ToBeDev := false;
              RemoveListItem
                   (slistptr,nextpitem^.info);
            end;
          if (nextpitem^.NbFathers = 0) then
            begin
              layerptr := GetLayer(PSync2,niv);
              RemoveFromTable(nextpitem);
              DeleteItem(layerptr,nextpitem);
            end;
        end;
    end
     { Included(nextpitem, res) }
  else
    begin
      newson := res;
      sonptr := father^.FirstSon;
      nb := 0;
      while sonptr <> nil do
        begin
          nb := nb+1;
          if sonptr^.Son^.Info = newson^.Info then
            itemarray[nb] := newson
          else
            itemarray[nb] := sonptr^.Son;
          sonptr := sonptr^.Next;
        end;
      if nextpitem^.ToBeDev = true then
        begin
          nextpitem^.ToBeDev := false;
          RemoveListItem(slistptr,nextpitem^.info);
        end;
      Backtracking(nextpitem,newson);
      father:=SearchTable(father^.Info,nb,itemarray);
```

```
                    end;
                  if (res^.NbFathers = 0) then
                    begin
                      layerptr := GetLayer(PSync2,niv);
                      RemoveFromTable(res);
                      DeleteItem(layerptr,res);
                    end;
                  newson^.ToBeDev := true;
                  AddListItem(slistptr,newson^.Info);
                end;
            end;
        end;
    end;
end;
{ Pour toutes les transitions "selfs" partant de s ...}
for j:=1 to nbSelfTransPoss do
  begin
    next := SelfTransPoss[j]^.destination;
    nextpitem := HasSon(father,next);
    { S'il n'existe pas encore de fils de valeur next ...}
    if nextpitem = nil then
      begin
       { Recherche d'un sous-AP existant }
       layerptr := GetLayer(PSync2,niv);
       nb := 0;
       son := s^.FirstSon;
       while son <> nil do
         begin
           nb := nb+1;
           itemarray[nb] := son^.Son;
           son := son^.Next;
         end;
       nextpitem := SearchTable(next,nb,itemarray);
       { Resultat nul }
       if nextpitem = nil then
         begin
           nextpitem := AddItem(layerptr,next);
           nextpitem^.First := false;
           RemoveFromTable(father);
           AddSon(father,nextpitem);
           AddDescenders(s,nextpitem);
           InsertInTable(nextpitem);
           InsertInTable(father);
           ItemNbElem(nextpitem);
           if nextpitem^.ToBeDev = false then
             begin
              nextpitem^.ToBeDev := true;
              AddListItem(slistptr,nextpitem^.Info);
             end;
         end
       { Resultat non nul }
       else
         begin
           { Recherche d'un nouveau pere }
           nb := 0;
           son := father^.FirstSon;
           inserted := false;
           while son <> nil do
             begin
              nb := nb+1;
              if (son^.Son^.Info > nextpitem^.Info) and
                 (not inserted) then
                begin
                  itemarray[nb] := nextpitem;
```

```
                inserted := true;
              end
            else
              begin
                itemarray[nb] := son^.Son;
                son := son^.Next;
              end;
          end;
        if not inserted then
          begin
           nb := nb+1;
           itemarray[nb] := nextpitem;
          end;
        newfather := SearchTable(father^.Info,nb,itemarray);
        { Nouveau pere trouve }
        if newfather <> nil then
          begin
           { BackTracking pour la substitution des deux peres }
           backtracking(father,newfather);
           father := newfather;
           nextpitem := HasSon(father,nextpitem^.Info);
           if nextpitem <> nil then
             if nextpitem^.ToBeDev = false then
               begin
                 nextpitem^.ToBeDev := true;
                 AddListItem(slistptr,nextpitem^.Info);
               end;
          end
          { Pas de nouveau pere }
        else
          begin
           RemoveFromTable(father);
           { Ajout de nextpitem comme fils }
           AddSon(father,nextpitem);
           InsertInTable(father);
           if nextpitem^.ToBeDev = false then
             begin
               nextpitem^.ToBeDev := true;
               AddListItem(slistptr, nextpitem^.Info);
             end;
          end;
      end;
  end
{ Father possede deja un fils de valeur next }
else
  if not(Included(s, nextpitem)) then
   begin
     { Recherche de l'union de s et nextpitem }
     newson := myunion(s,nextpitem,true);
     ItemNbElem(newson);
     { Si le noeud trouve est different de nextpitem,
       backtracking }
     if newson <> nextpitem then
       begin
         sonptr := father^.FirstSon;
         nb := 0;
         while sonptr <> nil do
          begin
            nb := nb+1;
            if sonptr^.Son^.Info = newson^.Info then
              itemarray[nb] := newson
            else
              itemarray[nb] := sonptr^.Son;
          end;
```

```
                    Backtracking(nextpitem,newson);
                    father := SearchTable(father^.Info,nb,itemarray);
                    if nextpitem^.NbFathers = 0 then
                     begin
                       RemoveFromTable(nextpitem);
                       if nextpitem^.ToBeDev then
                         RemoveListItem(slistptr,nextpitem^.info);
                       layerptr := GetLayer(PSync2,niv);
                       DeleteItem(layerptr,nextpitem);
                     end;
                  end;
                ItemNbElem(newson);
                ItemNbElem(father);
                if not newson^.ToBeDev then
                  begin
                    newson^.ToBeDev := TRUE;
                    AddListItem(slistptr,next);
                  end;
              end;
        end;
     end;

   end;

end;
```

2. Code de la procédure Transmit modifiée.

```
procedure Transmit(syncniv:integer;
                   sync:PTransition;source:PItem;
                   var res:PItem;resval:integer);

 var
   inserted : boolean;
   son : PSon;
   slistptr : PIntList;
   i,j,next : integer;
   resprim : PItem;
   rescree : boolean;

 begin
   slistptr := nil;
   rescree := false;
   { Niveau synchrone }
   if (niv+1) = syncniv then
     begin
       son := source^.FirstSon;
       while (son <> nil) do
        begin
          with automate[syncniv]^.Etat[son^.Son^.Info]^ do
            begin
              for j:=1 to nbSyncTransPoss do
               begin
                 if (SyncTransPoss[j]^.nom = sync^.nom) and
                    (SyncTransPoss[j]^.initiateur =
                                  (not (sync^.initiateur))) and
                    (SyncTransPoss[j]^.synchronisateur = sourceniv) then
                   begin
                    if opresult <> nil then
                     dispose (opresult);
                    {writeln('Caching au dernier niveau : ');}
                    if not RecTrans then
                     begin
                       new(opresult);
                       opresult^.item := nil;
                       opresult^.opOK := false;
                     end
                    else
                       opresult := GetOpResult(son^.Son,sync);

                    nextpitem := opresult^.item;
                    next := SyncTransPoss[j]^.destination;
                      { Pas de caching => }
                    if not opresult^.opOK then
                     begin
                       layerptr := GetLayer(PSync2,syncniv);
                       if rescree then
                         nextpitem := HasSon (res,next)
                       else
                         nextpitem := nil;
                       { res non cree ou pas de fils de valeur next }
                       if nextpitem = nil then
                         begin
                           nb := 0;
                           ltson := son^.Son^.FirstSon;
                           while ltson <> nil do
                            begin
                              nb := nb+1;
                              itemarray[nb] := ltson^.Son;
```

```
    ltson := ltson^.Next;
  end;
nextpitem := SearchTable(next,nb,itemarray);
{ Resultat de la recherche nul }
if nextpitem = nil then
 begin
   nextpitem := AddItem(layerptr,next);
   oldniv := niv;
   niv := syncniv;
   AddDescenders(son^.Son,nextpitem);
   ItemNbElem(nextpitem);
   niv := oldniv;
   nextpitem^.First := true;
   InsertInTable(nextpitem);
   if not rescree then
     begin
       layerptr := Getlayer(PSync2,niv);
       res := AddItem(layerptr,resval);
       rescree := true;
     end
   else
     RemoveFromTable(res);
   AddSon(res,nextpitem);
   InsertInTable(res);
   if not nextpitem^.ToBeDev then
     begin
       nextpitem^.ToBeDev := TRUE;
       AddListItem(slistptr,next);
     end;
 end
{ Resultat de la recherche non nul }
else
 begin
   if not rescree then
     begin
       nb := 1;
       itemarray[nb] := nextpitem;
     end
   else
     begin
       nb := 0;
       ltson := res^.FirstSon;
       inserted := false;
       while ltson <> nil do
        begin
          nb := nb+1;
          if (ltson^.Son^.Info>nextpitem^.Info)
             and (not inserted) then
            begin
              itemarray[nb] := nextpitem;
              inserted := true;
            end
          else
            begin
              itemarray[nb] := ltson^.Son;
              ltson := ltson^.Next;
            end;
        end;
       if not inserted then
        begin
          nb := nb+1;
          itemarray[nb] := nextpitem;
        end;
     end;
```

```
newres := SearchTable(resval,nb,itemarray);
{ Resultat de la recherche nul }
if newres = nil then
  begin
    if not rescree then
     begin
       layerptr := GetLayer(PSync2,niv);
       res := AddItem(layerptr,resval);
       rescree := true;
       AddSon(res,nextpitem);
       nextpitem^.First := true;
       InsertInTable(res);
       if not nextpitem^.ToBeDev then
         begin
           nextpitem^.ToBeDev := TRUE;
           AddListItem(slistptr,next);
         end;
     end
    else
     begin
       if res^.NbFathers = 0 then
         begin
           RemoveFromTable(res);
           AddSon(res,nextpitem);
           nextpitem^.First := true;
           InsertInTable(res);
           if not nextpitem^.ToBeDev then
            begin
              nextpitem^.ToBeDev := TRUE;
              AddListItem(slistptr,next);
            end;
         end
       else
         begin
           layerptr := GetLayer(Psync2,niv);
           res := AddItem(layerptr,resval);
           rescree := true;
           for i:=1 to nb do
            begin
              AddSon(res,itemarray[i]);
              itemarray[i]^.First := TRUE;
            end;
           InsertInTable(res);
           if not nextpitem^.ToBeDev then
            begin
              nextpitem^.ToBeDev := TRUE;
              AddListItem(slistptr,next);
            end;
         end;
     end;
  end
{ Resultat de la recherche non nul }
else
  begin
    if rescree then
     if res^.NbFathers = 0 then
       begin
         RemoveFromTable(res);
         layerptr := GetLayer(PSync2,niv);
         deleteitem(layerptr,res);
       end;
    res := newres;
    rescree := true;
    if not nextpitem^.ToBeDev then
```

```
            begin
              nextpitem^.ToBeDev := TRUE;
              AddListItem(slistptr,next);
            end;
        end;
    end;
  end
{ res cree et a un fils de valeur next }
else
  begin
    if res^.NbFathers = 0 then
     newson := myunion(son^.Son,nextpitem,true)
    else
     newson := myunion(son^.Son,nextpitem,false);
    ItemNbElem(newson);
    newson^.First := true;
    if newson <> nextpitem then
     begin
       nextpitem := newson;
       nb := 0;
       ltson := res^.FirstSon;
       while ltson <> nil do
         if ltson^.Son <> nextpitem then
           begin
            nb := nb+1;
            itemarray[nb] := ltson^.Son;
            ltson := ltson^.Next;
           end
         else
           begin
            nb := nb+1;
            itemarray[nb] := newson;
            ltson := ltson^.Next;
           end;
       newres := SearchTable(resval,nb,itemarray);
       { Resultat de la recherche nul }
       if newres <> nil then
         begin
           if res^.NbFathers = 0 then
            begin
              RemoveFromTable(res);
              layerptr := GetLayer(PSync2,niv);
              DeleteItem(layerptr,res);
            end;
           res := newres;
         end
       { Resultat de la recherche non nul }
       else
         begin
           if res^.NbFathers = 0 then
            begin
              RemoveFromTable(res);
              ReplaceSon(res,nextpitem,newson);
              InsertInTable(res);
              if nextpitem^.NbFathers = 0 then
                begin
                  RemoveFromTable(nextpitem);
                  layerptr:=
                      Getlayer(PSync2,syncniv);
                  DeleteItem(layerptr,nextpitem);
                end;
              if not newson^.ToBeDev then
                begin
                  newson^.ToBeDev := TRUE;
```

```
                    AddListItem(slistptr,next);
                  end;
               end
              else
               begin
                 layerptr := GetLayer(PSync2,niv);
                 newres := AddItem(layerptr,resval);
                 AddSon(newres,newson);
                 oldniv := niv;
                 niv := syncniv;
                 AddDescenders(res,newres);
                 niv := oldniv;
                 res := newres;
                 InsertInTable(res);
                 if not newson^.ToBeDev then
                   begin
                     newson^.ToBeDev := TRUE;
                     AddListItem(slistptr,next);
                   end;
               end;
            end;
        end;
    end;
   { Enregistrement de l'operation }
   if RecTrans then
     MyRecordOper(son^.Son,sync,nextpitem);
 end
{ Caching ...}
else
 { Si le resultat de la transition n'est pas nul }
 if nextpitem <> nil then
   begin
     if rescree then
       begin
        oldson := HasSon(res,nextpitem^.info);
        { Si res a un fils de valeur info }
        if oldson <> nil then
          begin
            if res^.NbFathers = 0 then
              newson := MyUnion(nextpitem,oldson,true)
            else
              newson:=MyUnion(nextpitem,oldson,false);
            nextpitem := newson;
            ItemNbElem(nextpitem);
            if newson <> oldson then
              begin
               nb := 0;
               ltson := res^.FirstSon;
               while ltson <> nil do
                 begin
                   nb := nb+1;
                   if ltson^.Son^.Info = oldson^.Info
                   then
                     itemarray[nb] := newson
                   else
                     itemarray[nb] := ltson^.Son;
                   ltson := ltson^.Next;
                 end;
               newres:=
                  SearchTable(res^.Info,nb,itemarray);
               { Resultat de la recherche non nul }
               if newres <> nil then
                 begin
                   if res^.NbFathers = 0 then
```

```
                begin
                 RemoveFromTable(res);
                 layerptr:= GetLayer(PSync2,niv);
                 DeleteItem(layerptr,res);
                end;
              res := newres;
            end
          { Resultat de la recherche nul }
          else
            begin
              RemoveFromTable(res);
              ReplaceSon(res,oldson,newson);
              InsertInTable(res);
            end;
        end;
    end
  { Si res n'a pas de fils de valeur info }
  else
    begin
      nb := 0;
      inserted := false;
      ltson := res^.FirstSon;
      while ltson <> nil do
        begin
         nb := nb+1;
         if (ltson^.Son^.Info > nextpitem^.Info)
            and (not inserted) then
           begin
             itemarray[nb] := nextpitem;
             inserted := true;
           end
         else
           begin
             itemarray[nb] := ltson^.Son;
             ltson := ltson^.Next;
           end;
        end;
      if not inserted then
        begin
         nb := nb+1;
         itemarray[nb] := nextpitem;
        end;
      newres:=
          SearchTable(res^.Info,nb,itemarray);
      { Resultat de la recherche non nul }
      if newres <> nil then
        begin
         if res^.NbFathers = 0 then
           begin
             RemoveFromTable(res);
             layerptr := GetLayer(PSync2,niv);
             DeleteItem(layerptr,res);
           end;
         res := newres;
        end
      { Resultat de la recherche nul }
      else
        begin
         RemoveFromTable(res);
         AddSon(res,nextpitem);
         InsertInTable(res);
        end;
    end;
end
```

```
                    { Res pas cree }
                    else
                      begin
                       nb := 1;
                       itemarray[nb] := nextpitem;
                       res := SearchTable(resval,nb,itemarray);
                       { Resultat de la recherche nul }
                       if res = nil then
                         begin
                           layerptr := GetLayer(PSync2,niv);
                           res := AddItem(layerptr,resval);
                           AddSon(res,nextpitem);
                           rescree := true;
                           InsertInTable(res);
                         end
                       { Resultat de la recherche non nul }
                       else
                           rescree := true;
                      end;
                    if not nextpitem^.ToBeDev then
                      begin
                       nextpitem^.ToBeDev := TRUE;
                       AddListItem(slistptr,next);
                      end;
                  end;
              end;
          end;
        son := son^.Next;
      end;
  end
{ Niveau intermediaire }
else
  begin
    son := source^.FirstSon;
    while son <> nil do
     begin
       layerptr := GetLayer(PSync2,niv+1);
       if opresult <> nil then
         dispose(opresult);
       if not RecTrans then
         begin
           new(opresult);
           opresult^.opOK := false;
           opresult^.item := nil;
         end
       else
           opresult := GetOpResult(son^.Son,sync);
       nextpitem := opresult^.item;
       { Si pas de caching }
       if not opresult^.opOK then
         begin
           { Appel de Transmit }
           niv := niv+1;
           transmit(syncniv,sync,son^.Son,resprim,son^.Son^.Info);
           niv := niv-1;
           { Enregistrement de l'operation }
           if RecTrans then
              MyRecordOper(son^.Son,sync,resprim);
           if resprim <> nil then
            begin
               ItemNbElem(resprim);
              if rescree then
                begin
```

```
      nb := 0;
      ltson := res^.FirstSon;
      inserted := false;
      while ltson <> nil do
       begin
         nb := nb+1;
         if (ltson^.Son^.Info > resprim^.Info) and
            (not inserted) then
           begin
             itemarray[nb] := resprim;
             inserted := true;
           end
         else
           begin
             itemarray[nb] := ltson^.Son;
             ltson := ltson^.Next;
           end;
       end;
      if not inserted then
       begin
         nb := nb+1;
         itemarray[nb] := resprim;
       end;
    end
  else
    begin
      nb := 1;
      itemarray[nb] := resprim;
    end;
  newres := SearchTable(resval,nb,itemarray);
  if newres <> nil then
    begin
      if rescree then
       if res^.NbFathers = 0 then
         begin
           RemoveFromTable(res);
           layerptr := GetLayer(PSync2,niv);
           DeleteItem(layerptr,res);
         end;
      res := newres;
      rescree := TRUE;
    end
  else
    if rescree then
      if res^.NbFathers = 0 then
       begin
         RemoveFromTable(res);
         AddSon(res,resprim);
         layerptr := GetLayer(PSync2,niv);
         LayerNbElem(layerptr);
         InsertInTable(res);
       end
      else
       begin
         layerptr := GetLayer(PSync2,niv);
         newres := AddItem(layerptr,resval);
         for i:= 1 to nb do
           begin
             AddSon(newres,itemarray[i]);
             itemarray[i]^.First := TRUE;
           end;
         res := newres;
         rescree := true;
         InsertInTable(res);
```

```
            end
          else
            begin
             layerptr := GetLayer(PSync2,niv);
             res := AddItem(layerptr,resval);
             rescree := true;
             AddSon(res,resprim);
             LayerNbElem(layerptr);
             InsertInTable(res);
            end;
        AddListItem(slistptr,resprim^.Info);
        resprim^.ToBeDev := TRUE;
      end;
  end
{ Caching }
else
  if nextpitem <> nil then
    begin
     if rescree then
       begin
         oldson := HasSon(res,nextpitem^.info);
         if oldson <> nil then
           begin
            if res^.NbFathers = 0 then
              newson := MyUnion(nextpitem,oldson,true)
            else
              newson := MyUnion(nextpitem,oldson,false);
            nextpitem := newson;
            ItemNbElem(nextpitem);
            if nextpitem <> oldson then
              begin
                nb := 0;
                ltson := res^.FirstSon;
                while ltson <> nil do
                  begin
                   nb := nb+1;
                   if ltson^.Son^.Info = nextpitem^.Info then
                     itemarray[nb] := nextpitem
                   else
                     itemarray[nb] := ltson^.Son;
                   ltson := ltson^.Next;
                  end;
                newres := SearchTable(res^.Info,nb,itemarray);
                if newres <> nil then
                  begin
                   if res^.NbFathers = 0 then
                     begin
                       RemoveFromTable(res);
                       layerptr := GetLayer(PSync2,niv);
                       DeleteItem(layerptr,res);
                     end;
                   res := newres;
                  end
                else
                  begin
                   RemoveFromTable(res);
                   ReplaceSon(res,oldson,nextpitem);
                   InsertInTable(res);
                  end;
              end;
           end
         else
           begin
            nb := 0;
```

```
        inserted := false;
        ltson := res^.FirstSon;
        while ltson <> nil do
          begin
            nb := nb+1;
            if (ltson^.Son^.Info > nextpitem^.Info) and
              (not inserted) then
              begin
               itemarray[nb] := nextpitem;
               inserted := true;
              end
            else
              begin
               itemarray[nb] := ltson^.Son;
               ltson := ltson^.Next;
              end;
          end;
        if not inserted then
          begin
            nb := nb+1;
            itemarray[nb] := nextpitem;
          end;
        newres := SearchTable(res^.Info,nb,itemarray);
        if newres <> nil then
          begin
            if res^.NbFathers = 0 then
              begin
               RemoveFromTable(res);
               layerptr := GetLayer(PSync2,niv);
               DeleteItem(layerptr,res);
              end;
            res := newres;
          end
        else
          begin
            RemoveFromTable(res);
            AddSon(res,nextpitem);
            InsertInTable(res);
          end;
      end;
  end
else
  begin
    nb := 1;
    itemarray[nb] := nextpitem;
    res := SearchTable(resval,nb,itemarray);
    if res = nil then
      begin
       ItemNbElem(nextpitem);
       layerptr := GetLayer(PSync2,niv);
       res := AddItem(layerptr,resval);
       AddSon(res,nextpitem);
       rescree := true;
       InsertInTable(res);
      end
    else
      rescree := true;
  end;
if not nextpitem^.ToBeDev then
  begin
    nextpitem^.ToBeDev := TRUE;
    AddListItem(slistptr,nextpitem^.info);
  end;
ItemNbElem(res);
```

```
                layerptr := GetLayer(PSync2,niv);
                LayerNbElem(layerptr);
              end;
          son := son^.Next;
        end;
    end;
  if rescree then
    if (res^.FirstSon <> nil) then
      begin
       niv := niv+1;
       Propagate(slistptr,res);
       niv := niv-1;
      end
    else
      begin
       RemoveFromTable(res);
       layerptr := GetLayer(PSync2,niv);
       DeleteItem(layerptr,res);
       res := nil;
      end
 else
   res := nil;
end;
```